University of California
Santa Cruz

Distributed Routing for Very Large Networks Based on Link Vectors

A dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy
in
Computer Science
by
Jochen Behrens
June 1997

The dissertation of Jochen Behrens is approved:

Prof. J. J. Garcia-Luna-Aceves

Prof. Anujan Varma

Prof. Allen Van Gelder

Dean of Graduate Studies and Research

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUN 1997		**00-06-1997 to 00-06-1997**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Distributed Routing for Very Large Networks Based on Link Vectors	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**119**	

Contents

List of Figures

Distributed Routing for Very Large Networks Based on Link Vectors

Jochen Behrens

Abstract

Routing is the network-layer function that selects the paths that data packets travel from a source to a destination in a computer communication network. This thesis is on distributed adaptive routing algorithms for large packet-switched networks.

A new type of routing algorithms for computer networks, the link-vector algorithm (LVA) is introduced. LVAs use selective dissemination of topology information. Each router running an maintains a subset of the topology that corresponds to adjacent links and those links used by its neighbor routers in their preferred paths to known destinations. Based on that subset of topology information, the router derives its own preferred paths and communicates the corresponding link-state information to its neighbors. An update message contains a vector of updates; each such update specifies a link and its parameters. LVAs can be used for different types of routing policies. LVAs are shown to have better performance than the ideal link-state algorithm based on flooding and the distributed Bellman-Ford algorithm.

Like other routing algorithms based on the distribution of link-state information, LVAs rely on sequence numbers to validate information that a router receives. A fundamental problem is to bound the sequence-number space. A new sequence-number reset algorithm is presented that uses a recursive query-response procedure and is designed to eliminate the need for periodic transmissions of link-state updates and aging. This new reset algorithm is applicable to routing protocols based on both flooding and selective distribution of link-state information.

An area-based link-vector algorithm (ALVA) is introduced for the distributed maintenance of routing information in very large internetworks. According to ALVA, destinations

in an internetwork are aggregated in areas in multiple levels of hierarchy. Routers maintain a database that contains a subset of the topology at each level of the hierarchy. Advantages of ALVA over existing hierarchical protocols based on link-states are that ALVA does not require backbones and accommodates multiple hierarchy levels.

Acknowledgments

Many people accompanied me on my way through graduate school and helped me meet the challenges that I encountered during the last four and a half years here in Santa Cruz.

No one deserves more grateful recognition than my wife, Karin. Without her sacrifices, love, and moral support, work would have been so much harder and life so much less meaningful.

J.J. Garcia-Luna-Aceves was the best adviser one could wish for. His support, patience, and guidance made this work possible. He had the original idea to use link-states with partial topology information. I feel very fortunate to have been given the chance to be one of the first members of his research group here at UCSC. Working with him and in his group was such a pleasure. Thanks, J.J.!

I would like to thank Allen Van Gelder for his support early in my career at UCSC, and for serving on my thesis committee. Thanks also to Anujan Varma, for his support and for serving on my committee. In his great lectures Anujan also taught me a lot of what I know about computer networks.

A big thank you to all the members of the coco-group. Working with you was a lot of fun. In particular, Shree was the perfect "secretary," and Chane was a great friend in and out of the lab.

Jorge was always around as a friend and helped us to explore the natural beauty of California. And when it came to fighting the bears or preparing quesadillas in the wilderness, he was simply the best.

I am also grateful for all the stimulating discussions and the good times I experienced with Paul, Bruce, Rob, Mike, Leslie, Lampros, Christos, Dimitrios, Melissa, Mumu, and all the other people whose names elude my mind right now.

Thanks also to the staff and faculty of the Baskin Center, who made up such a great environment. Lynne Sheehan stood out with her efforts to pave the road of her graduate students.

Last but not least, I would like to thank my parents for all their love and support, and my parents-in-law for gracefully suffering through my taking their daughter (and their future grandchild) so far away.

The text of this dissertation includes material that has previously been published in [BG94], [GB95], and [BG97]. The co-author listed in these publications directed and supervised the research which forms the basis for this dissertation.

This work was supported in part by the Office of Naval Research (ONR) under Contract No. N-00014-92-J-1807 and by the Defense Advanced Research Projects Agency (DARPA) under contracts F19628-93-C-0175 and F19628-96-C-0038.

Chapter 1

Introduction

An internetwork consists of a collection of interconnected domains, where each domain is a collection of such resources as networks, routers, and hosts, under the control of a single administration. Routing is the network-layer function that selects the paths that data packets travel from a source to a destination.

Todays internetworks are based on store-and-forward or packet switching. When a data packet arrives at a router, the router decides on which output port it needs to be transmitted. In virtual circuit switching, a path is set up when a session is initiated and maintained during the life of the session; all data travels along this pre-established path. When datagram routing is used, each packet finds its own way through the network; successive packets may follow different routes [BG92].

The purpose of the routing algorithms is to set up hop-by-hop forwarding tables or *routing tables* at the routers. These tables specify the neighbor to which a data packet is forwarded towards its destination. It is essential for the operation of networks that entries in these routing tables do not contain persistent forwarding loops.

Routing algorithms can work either centrally or in a distributed fashion; routes can be set deterministically or adaptively. In a very large network, central control is not desirable because of high communication cost and the dependence on the reliability of a central unit. A routing protocol should be adaptive to handle resource failures and additions, and to

cope with cost changes caused by congestion, for example. For these reasons, this thesis focuses on distributed, adaptive routing protocols.

Other important design goals for routing protocols are optimality, simplicity, efficiency, and robustness. Routing algorithms should be able to optimize their performance according to the objectives of the designer and users. They should be as simple as possible, but must make efficient use of processing and memory resources, and utilize a minimum of the communication capacities. Routing algorithms should perform correctly when faced with unforeseen circumstances and prove stable under a variety of network conditions [Cis97, Mur96].

Cost metrics for links are chosen to determine paths through the network that are optimized according to policies set by network administrators. Metrics can be static or dynamic. The simplest static cost metric is to assign unit cost to every link, the resulting routing policy is minimum-hop routing. Cost metrics can also be based on various characteristics of the link, such as bandwidth, delay, reliability or communication cost. Dynamic cost metrics allow the network to react, for example, to congestion by taking into account the time packets are queued up in buffers. However, it is not desirable that link costs change too frequently, because this would lead to excessive routing overhead and may cause oscillatory behavior. As an example, the cost metric used in the ARPANET is based on a hop-normalized-delay function [KZ89, SADM95]. Cisco's IGRP and EIGRP protocols use a combination of internetwork delay, bandwidth, reliability, and load as its link metric [Cis97].

For quality of service routing, other parameters, such as the delay jitter or bandwidth guarantees will need to be taken into account. The implications that the choice of link cost metrics has for the performance of internetworks are beyond the scope of this thesis; we simply assume that such a cost is given and that it is determined by an underlying protocol.

All the work in inter-domain and intra-domain routing for data networks has proceeded in two main directions: distance-vector protocols in which routers exchange vectors of distances of preferred paths to known destinations, and link-state protocols in which routers replicate complete topology information with which they compute their preferred paths.

Most DVAs are based on the distributed Bellman-Ford algorithm (DBF) [BG92]. In essence, nodes exchange information about the length of their paths to destinations; this information, together with the length of adjacent links, is used locally to compute the best path.

In the distributed Bellman-Ford algorithm, for a given destination i, each node keeps its best known distance D_i to the destination in its memory. Initially, $D_i = 0$ at node i, and $D_i = \infty$ at all other nodes. In addition, a node j knows the cost d_{jk} of its link to any neighbor k. Let $D_i^j(t)$ denote the distance to i as known by node j at time t, and $D_{ki}^j(t)$ the distance from k to i as reported to j by its neighbor k at time t. When node j receives an update message from k, it updates its distance according to the iteration

$$D_i^j(t+1) = \min(D_i^j(t), D_{ki}^j(t) + d_{jk})$$

$$D_i^i(t+1) = 0.$$

Either periodically or when its distance changed, a node sends an update to its neighbors. This algorithm terminates within a finite amount of time with the correct shortest distance to i known at every node [BT89].

To be able to react to changes in the topology, nodes store the distances as reported by their neighbors. When a neighbor reports a distance different from the one stored already, node j will choose its new distance to i to be

$$D_i^j = \min_{k \in N_j} (D_{ki}^j + d_{jk}),$$

where N_j is the set of neighbors of j. The neighbor through which this shortest path goes is chosen as the next-hop entry in the routing table.

The two main problems to be solved in protocols based on distance vectors are the possibility of loops and DBF's counting-to-infinity. Figure 1.1 illustrates the counting-to-infinity problem of DBF. Consider the very simple topology in Figure 1.1. Node k has a distance of 1 to i, node j a distance of 2. When link (k, i) fails, node k chooses its new

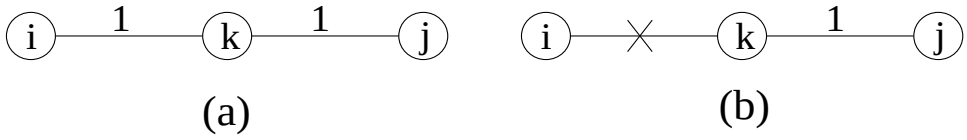

FIGURE 1.1: Example for Counting-to-Infinity

minimum distance to be 3, the length of (k, j) plus j's reported distance of 2, and reports this to j. j in turn adjusts its distance to 4 and reports this to k. This process continues until a maximum distance is reached that indicates that the link has failed. While this process goes on, k's next hop toward i is j, while j attempts to route through k, creating a loop in the forwarding tables.

Many mechanisms have been proposed to solve these problems. According to the *Split Horizon* technique, routers do not advertise paths to the neighbor from which they received them. The *Poison Reverse* technique goes one step further and advertises infinite cost to destinations received by the neighbor. While these techniques do eliminate many of the simple cases in which counting-to-infinity occurs, they are not always effective [Tho96, Hui95].

In other approaches, additional information about the links is propagated to maintain the spanning tree of the neighboring nodes, or a synchronizing mechanism is used to ensure that for each destination, the routing table entries of all routers define a directed acyclic graph at all times [Gar93].

Path-finding algorithms exchange distance vectors containing the length and the second-to-last hop in that path [CRKG89, Gar86, Hum91, RF91, Mur96, GM97]. For any destination in an update message, the predecessor router or network is indicated in addition to the length of the path. This information implicitly contains the complete spanning trees used by a router; by following the predecessor information, the complete path to any destination can be extracted. Using this information, routing decisions that would lead to long term looping can be avoided.

Path-vector algorithms specify complete path information for any destination they need to reach [LR91, Rek93]. With this information, a router can determine whether the paths

to a destination advertised by different neighboring routers are contradictory, whether they may lead to oscillating behavior or to forwarding loops.

Examples for protocols based on DVAs are BGP [LR91], IDRP [ISO], RIP [Hed88], Cisco's IGRP [Bos92], and EIGRP [AGB94].

The key advantage of protocols based on distance-vector algorithms (DVA) is that they scale well for a given combination of services taken into account in a cost metric. Because route computation is done distributedly, DVAs are ideal to support the aggregation of destinations to reduce communication, processing, or storage overhead. However, an inherent limitation of DVAs is that routers exchange information regarding path characteristics, not link or node characteristics. Because of this, the storage and communication requirements of *any* DVA grows proportionally to the number of combinations of service types or policies [Jaf84].

Link-state algorithms (LSA) are based on the flooding of link information (topology broadcast). At every node, a topology database containing information about every link is maintained. To flood the network with link state information, the source of the information broadcasts it to all it neighbors. A node that receives new information in turn forwards it to all its neighbors. Thus, the complete topology information is replicated at every node and then used to compute the routing tables.

Because protocols based on link-state algorithms (LSA) keep complete topology information at routers, they avoid the long-term looping problems of old distance-vector protocols (e.g., RIP). While routers may have inconsistent views of the topology that can cause loops in the forwarding tables, these loops are short lived because they persist at most for the time it takes updates to reach all destinations. More importantly, an LSA exchanges information regarding link characteristics, which means that the complexity of storing and disseminating routing information to support multiple types of services and policies grows linearly with the service types and policies, not their combinations. However, the requirement that the complete topology be broadcasted to every router does not scale well [ERH92]. There are two main scaling problems: flooding requires excessive communication resources,

and computing the routes using complete topology databases requires excessive processing resources.

Examples for protocols based on LSAs are the inter-domain policy routing (IDPR) architecture [EST93], ISO IS-IS [ISO89], and OSPF [Moy94].

To cope with the inherent overhead of flooding, today's LSAs organize the network or internet into areas connected by a backbone. However, this imposes additional network configuration problems and, as results obtained by Garcia-Luna-Aceves and Zaumen indicate [GZ94], areas must be chosen carefully, together with the masks used to hide information regarding destinations in an area, for any performance improvement to be obtained with respect to a "flat" LSA. Furthermore, aggregating information in an LSA is much more difficult than in a DVA (e.g., see [Gar88, RHE94]). Because LSAs require topology maps to be replicated at each router, different levels of topologies must be defined and routers must use multiple topology maps to aggregate information in an LSA (e.g., [Moy94, TRTN89]). In contrast, aggregating information in DVAs is very simple, because DVA exchanges information about distances to destinations, and such destinations can be a single network entity (e.g., router, network, host) or a group of entities (e.g., areas, confederations, clusters). Accordingly, the routing algorithm uses a single routing table with entries to individual or aggregated destinations [McQ74].

In summary, the routing algorithms used in todays' routing protocols have severe scaling problems. The dissemination of information on a per path basis in DVAs leads to a combinatorial explosion with the number of service types and policies. In LSAs, the replication of the complete topology information consumes excessive communication and processing resources in very large networks. In spite of these inherent problems, all of the existing routing protocols and proposals for protocols for very large networks are based on either of the two algorithm types.

To address these problems, we describe a new method for distributed, adaptive routing: *link-vector algorithms* (LVA). The key idea of LVAs is that each router reports to its neighbors the characteristics of those links that it uses in its preferred paths. Using this

information, each router constructs a *source graph* that consists of a partial topology and is used to compute its preferred paths. By disseminating link-states rather than path characteristics, LVAs avoid the combinatorial explosion incurred with DVAs for routing under multiple constraints. At the same time, because only relevant information is propagated, the communication and storage overhead can be significantly less when compared to flooding based LSAs. The rest of this thesis is organized as follows:

Chapter 2 presents the link-vector algorithm (LVA,) describes its operation, points out differences to other routing algorithms, verifies the correctness of LVA, and analyzes its performance.

Chapter 3 introduces and verifies a novel mechanism to verify updates for protocols that rely on the dissemination of link-state information.

Chapter 4 describes the extension of LVA to an area-based hierarchical routing architecture.

Chapter 5 summarizes the work presented in this thesis and points at some directions for future work.

Chapter 2

Link-Vector Algorithm (LVA)

Although the inherent limitations of LSAs and DVAs are well known, existing routing protocols or proposals for routing in large internets are based on these two approaches (e.g., see [CCS95, EST93]). This chapter presents a new method for distributed, scalable routing in computer networks called *link vector algorithms*, or LVA. The basic idea of LVA consists of asking each router to report to its neighbors the characteristics of each of the links it uses to reach a destination through one or more preferred paths, and to report to its neighbors which links it has erased from its preferred paths. Using this information, each router constructs a *source graph* consisting of all the links it uses in the preferred paths to each destination. LVA ensures that the link-state information maintained at each router corresponds to the link constituency of the preferred paths used by routers in the network or internet. Each router runs a local path-selection algorithm or multiple algorithms on its topology table to compute its source graph with the preferred paths to each destination. Such path-selection algorithm can choose any type of path (e.g., shortest path, maximum-capacity path, policy path.) The only requirements for correct operation are for all routers to use deterministic algorithms that produce the same result when computing the preferred paths (e.g., one router can use Bellman-Ford also to compute shortest paths from its source graph, while others can use Dijkstra's algorithm,) that the same tie-breaking rules are used for equally good paths, and that routers report all the links used in all preferred paths obtained using their local algorithm.

Because LVAs propagate link-state information by diffusing link states selectively based on the distributed computation of preferred paths, LVAs reduce the communication overhead incurred in traditional LSAs, which rely on flooding of link states. Because LVAs exchange routing information that is related to link (and even node) characteristics, rather than path characteristics, this approach eliminates the complexity incurred with DVAs for routing under multiple constraints [Jaf84]. Regardless of the type of routing algorithm used, aggregation information (i.e., hierarchical routing) becomes a necessity to support routing in very large networks or internets. LVAs report links needed to reach destinations, not complete topology maps, and a destination can be a single entity or an aggregate of entities. Therefore, aggregation of information can take place in an LVA by adapting any of the area-based routing techniques proposed for DVAs in the past (e.g., [Gar88, KK77, Tsu88]).

The following sections introduce the network model used in the remainder of this thesis, describe LVA, show that LVA converges to correct paths a finite time after the occurrence of an arbitrary sequence of link-cost or topological changes under the assumption that all routers run the same local algorithm(s) for the computation of preferred paths, calculate its complexity, and compare its average performance against the performance of an ideal LSA and the Distributed Bellman Ford (DBF) algorithm used in many DVAs.

2.1 Network Model

To describe the routing algorithms introduced in this thesis, an internet is modeled as a directed connected graph $G = (V, E)$, where V is the set of nodes and E the set of edges. An edge between nodes i and j is denoted as (i, j). Routers are the nodes of the graph and networks or direct links between routers are the edges of the graph. Each point-to-point link in such a graph has length or cost associated with it. In terms of connectivity, the graph is symmetrical. This means that if a link is operational in one direction, then it is operational in the opposite direction, too. A multipoint link is represented in the graph as a clique of all nodes that are connected by it.

An underlying protocol assures that

- A node detects within a finite time the existence of a new neighbor or the loss of connectivity with a neighbor.
- All messages transmitted over an operational link are received correctly and in the proper sequence (i.e., in the order that they were sent) within a finite time.
- All messages, changes in the cost of a link, link failures, and new-neighbor notifications are processed one at a time within a finite time and in the order in which they are detected.

Each router has a unique identifier, and link costs can vary in time but are always positive. Furthermore, routers are assumed to operate correctly, and information is assumed to be stored without errors. The same model can be applied to a single computer network.

The failure of a node is modeled as the failure of all links adjacent to that node.

2.2 Description of the Algorithm

The basic idea of LVA consists of asking each router to report to its neighbors the characteristics of every link it uses to reach a destination through a preferred path. The set of links used by a router in its preferred paths is called the *source graph* of the router. The topology known to a router consists of its adjacent links and the source graphs reported by its neighbors. The router uses this topology information to generate its own source graph using one or more local algorithms, which we call *path-selection algorithms*. A router derives a routing table specifying the successor, successors, or paths to each destination by running local algorithms on its source graph that can, of course, be part of the path-selection algorithms.

Figure 2.1(a) shows an example topology in which each link has the same cost in both directions. Figures 2.1 (b) through (e) show the selected topology known according to LVA with shortest-path routing at the routers highlighted in black. Solid lines represent the

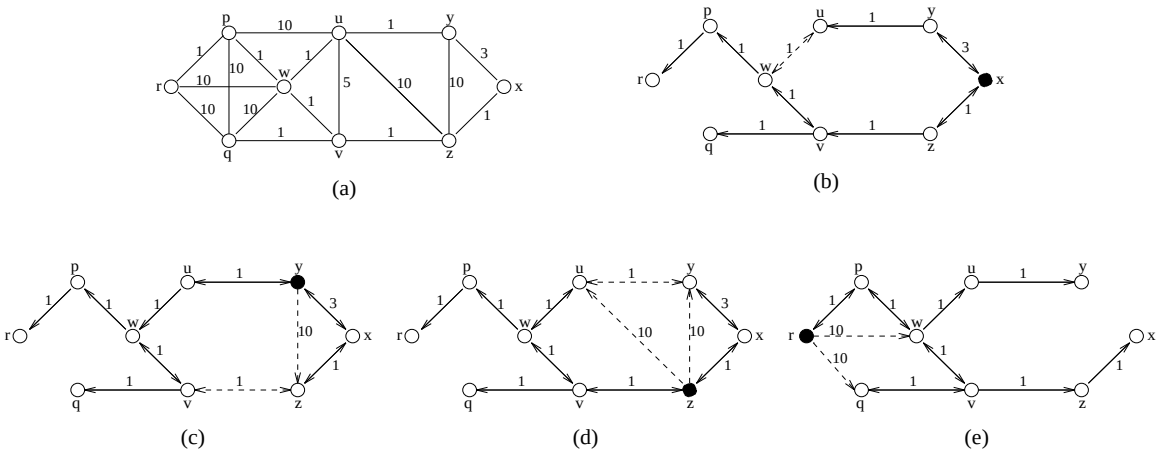

FIGURE 2.1: Example topology. Topology as seen by nodes indicated with filled circle. Solid lines indicate links in source graph; dashed lines indicate links in topology table but not in source graph; arrowheads indicate the stored direction.

links that are part of the source graph of the respective router, dashed links represent links that are part of the router's topology table but not of its source graph. Arrowheads on links indicate the direction of the link stored in the router's topology table. A link with two arrowheads corresponds to two links in the topology table; since the source graph is a tree rooted at the black node in the case of shortest-path routing, only the direction pointing away from the black node can be part of the source graph. Router x's source graph shown in Figure 2.1(b) is formed by the source graphs reported by its neighbors y and z (these are formed by the links in solid lines shown in Figures 2.1(c) and (d)) and the links for which router x is the head node (namely links (x, y) and (x, z)). A router's topology table may contain a link in only one direction (e.g., link (y, u) in Figure 2.1(b)); this is because a router's source graph contains links only in the directions of its preferred paths.

In addition to the parameters of a link, the record of each link entry in the topology table contains the set of neighbors that reported the link, and control information used to detect the validity of updates received for that particular link.

The basic update unit in LVA is a link-state update reporting the characteristics of a link; an update message contains one or more updates. For a link between router x and router or destination y, router x is called the *head node* of the link in the x to y direction.

For a multi-point link, a single head node is defined. The head node of a link is the only router that can report changes in the parameters of that link.

The main complexity in designing LVA stems from the fact that routers with different topology databases can generate long-term or even permanent routing loops if the information in those databases is inconsistent on a long-term or permanent basis.

Sending updates specifying only those links that a router currently uses in its preferred paths is not sufficient in LVA, because a given router sends incremental updates and may stop forwarding state information regarding one or more links that are not changing the values of their parameters. When this happens, it is not possible to ascertain whether the router is still using those links in preferred paths if routers' updates specify only those links currently used in preferred paths. Simply aging link-state information would lead to unnecessary additional control traffic and routing loops, especially in very large internets. Therefore, to eliminate long-term or permanent routing loops, routers must not only tell its neighbor routers which links they use in their preferred paths, but also which links they no longer use. Accordingly, routers using LVA send update messages with two types of update entries: *add* updates and *delete* updates. An *add* update reports a link that has been added to the source graph of the sending router or whose information has been updated; a *delete* update specifies a link that has been deleted from the source graph of the sending router. An update specifies all the parameters of the link (just like in an LSA.) A router sends an update in a message only when a link is modified, added, or deleted in its source graph, not when the same unmodified link is used for a modified set of preferred paths; therefore, the number of update messages and the size of update messages do not necessarily increase with the number of paths, service types, or policies that a router uses.

A router reports its source graph to its neighbors incrementally; therefore, a typical update message in LVA contains only a few *add* and *delete* updates. Of course, when a router establishes a new link, it has to send its entire source graph to the new neighbor; this is equivalent to the LSA case in which a router sends its entire topology table to a new

neighbor, or the DVA case in which a router sends its entire routing table to a newly found neighbor.

Because of delays in the routers and links of the internet, the *add* or *delete* updates sent by a router may propagate at different speeds along different paths. Therefore, a given router may receive an update from a neighbor with stale link-state information. The consistency of link-state information can be controlled on a link-by-link basis taking advantage of the fact that the only router that can change the information about a given link is its head node. More specifically, a distributed termination-detection mechanism is necessary for a router to ascertain when a given update is valid and avoid the possibility of updates circulating in the network forever [BG92, BG94]. Termination-detection mechanisms based on sequence numbers similar to those used in a number of LSAs [BG92, Moy94, Per83] or diffusing computations [Gar92] can be used to validate updates.

A concrete embodiment of LVA for shortest-path routing, which we call LVA-SEN, is shown in the appendix of this chapter (Section 2.8. LVA-SEN determines the validity of updates using a sequence number for each link. The sequence number of a link consists of a counter that can be incremented only by the head node of the link. For convenience, a router is assumed to keep only one counter for all the links for which it is the head node, which simply means that the sequence number a router gives to a link for which it is the head node can be incremented by more than one each time the link parameters change values. The information regarding each link in a router's topology table is augmented to include the sequence number of the most recent update received, which was generated by the head node of that link.

For simplicity, the specification of LVA-SEN assumes that unbounded counters are used to keep track of sequence numbers and that each router remembers the sequence number of links deleted from its topology table long enough for the algorithm to work correctly. In practice, if finite sequence numbers are used to validate updates, a reset mechanism is needed to recycle sequence numbers. An age field serves this purpose [BG92]; when a link is deleted from the topology table, the router maintaining the table labels the link as deleted,

1. Check whether the update is valid.
2. If the update is valid, two cases need to be distinguished:
   (a) ADD update: add the link to the topology table, or, if the link is already in the table, add the sender of the packet to the set of reporting nodes.
   (b) DELETE update: remove the sender of the message from the set of reporting nodes; if the set is empty, remove the link from the topology table.
3. Run the path selection algorithm to construct the new source graph.
4. Construct the new routing table from the source graph.
5. Compare the old and new source graphs:
   - Produce an ADD update for links that are in the new but not in the old source graph, and for those links that changed.
   - Produce a DELETE update for links that are no longer used (i.e. they are in the old but not in the new source graph.
6. Send a message with the updates produced in the previous step to all neighbors.

FIGURE 2.2: Basic operation of LVA. The first two steps are performed for every update in the message. The remaining steps are performed once the topology table has been updated.

which means that it is not used to compute new source graphs, and keeps the link entry until the age field of the entry expires. To limit the size of the age field, the head node of each link sends an *add* or *delete* update for that link periodically (depending on whether or not the link is in its source graph), even if no changes occur in the link. The maximum age of a link is then a multiple of the time it takes for an update to propagate throughout the network.

Procedures *update*, *update_topology_table*, and *compare_source_graphs* are the core of LVA-SEN, in that these procedures are performed to update the data structures held at the router each time a router processes an input event (e.g., a message from one of its neighbors, or a link-cost change notification from an underlying protocol). Figure 2.2 summarizes the operation of these procedures in six steps.

In LVA-SEN, when a router processes an *add* or *delete* update, it first compares the sequence number in that update against the sequence number maintained for the same link in the topology table. The update is processed if either it specifies a larger sequence number than the one stored in the topology table, or no entry for the link exists in the topology

table. In the case of an *add* update, the link state is added to the topology table, or the new values of the link parameters replace the current entry, or the reporting node is added to the set of nodes that reported that link. For the case of a *delete* update, if there is an entry concerning the reported link in the topology table, then the reporting node is removed from the set of reporting nodes, and the link is deleted from the topology table if that set becomes empty and the link is not an outgoing link of finite length. An update is discarded if it specifies a sequence number that is smaller than the one in the topology table. In this case, the receiving node prepares an update for the same link intended to correct the information stored by the neighbor that sent the update.

Dijkstra's algorithm [BG92], or any other shortest-path algorithm, is run on the updated topology table to construct a new source graph, which constitutes a shortest-path tree; in this case, the new routing table is generated together with the source graph. The router compares the new source graph against its old one (procedure *compare_source_graphs*), and an update message is sent to all neighbors with the differences between the two. In addition to changes in membership in the source graph, a link in the source graph is considered different if its sequence number is changed. If the different link is in the new source graph, then an *add* update about this link is added to the update message. If the link is in the old source graph but is not in the new one, then a *delete* update is added. If any of the link entries refer to the state of an outgoing link of the node itself, then it gets a current sequence number. When a router sends an update message, it increments its sequence-number counter and discards its old source graph.

If a link cost changes, then its head node is notified by an underlying protocol. The node then runs *update* with the appropriate message as input. This holds for simple changes in link cost, as well as for a link failure. In the latter case, the link cost is set to infinity. The same approach is used for a new link or a link that comes up again after a failure. In the case of a failing node, all its neighbors are notified about the failure of their links to the failed node; these nodes then remove the failed node from the list of reporting nodes for all affected links, and therefore obtain an accurate picture of the topology after running

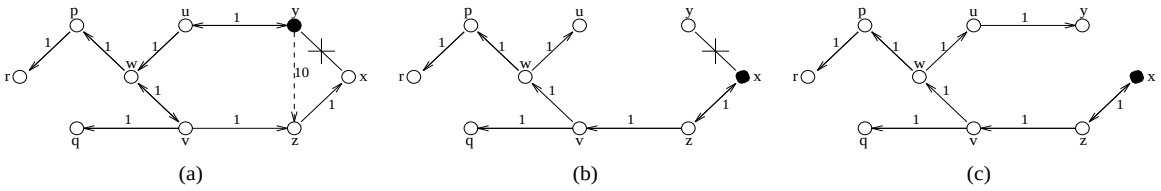

FIGURE 2.3: View of topology at nodes x and y after link (x, y) fails

the *update* procedure. Multiple changes in the status of nodes or links may be implied by the input event (e.g., a message received, a link failure). A node processes all such changes before it sends its own update message.

We assume that the node that comes up for the first time after a failure does not 'remember' any information that it previously had; in particular, it does not know the last sequence number it used. After initializing its data structures, the node that comes up sends a query to all its neighbors. In response, its neighbors send back their complete source graphs, plus the latest sequence number they received from the node (nodes store sequence numbers of neighboring nodes, which are updated when a link of some neighbor is changed). The node collects all this information, updates its topology table and sequence number, and then performs the same steps as in the procedure *update*.

Consider the topology of Figure 2.1 and assume that link (y, z) fails. Nodes y and z process this failure and call procedure *link_down*; because neither node uses the failed link in its source graph, no update results from this failure. Consider now the case in which link (x, y) fails. Nodes x and y call procedure *link_down*. Because link (x, y) is used in the source graph of both nodes, they must send an update message. The message sent by node y to its neighbors specifies a *delete* update for link (y, x) and *add* updates for links (v, z) and (z, x), which must now be used to reach nodes z and x, respectively, and, therefore, must be added to node y's source graph; furthermore, its update message also specifies delete updates for link (x, z) which cannot be used to reach node z anymore. Figure 2.3(a) shows node y's view of the topology after the link failure. If node u used the path u, v, z, x to reach node x, the update from node y causes no changes to u's source graph and it simply updates its topology table based on y's update.

On the other hand, when node x processes the failure of link (x, y), it has no path left to reach node y, because it has no link incident into node y left in its topology table. Node x sends a message containing *delete* updates for links (x, y) and (y, u), and an *add* update for link (w, u). Figure 2.3(b) shows node x's view of the topology at that time. When node z processes node x's update, it sends and update message that must contain an *add* update for link (u, y) and a *delete* update for link (x, y). When node x processes that message, it obtains the view of the topology shown in Figure 2.3(c). Note that node z's update does not create any changes in node v's source graph, who must reach node y through path v, w, u, y; therefore, node v does not send any update as a result.

2.3 Differences With Previous Methods

Three types of prior algorithms have been or can be used to compute preferred paths based on link-state information: link-state algorithms (LSA), *path-finding algorithms* (PFA), and *path-vector algorithms* (PVA).

2.3.1 Differences with LSAs

LSAs are also called *topology broadcast algorithms*. In an LSA, information about the state of each link in the network is sent to every router by means of a reliable broadcast mechanism, and each router uses a local algorithm to compute preferred paths. The key difference between LSAs and LVAs is that each link-state update propagates to all routers in an LSA, while in LVA the update propagates to only those routers that use the corresponding link in a path to a destination and their neighbors. Therefore, the reliable broadcast mechanisms used in LSAs to ensure that all routers with a physical path to a source of link-state updates receives the most recent updates within a finite time (e.g., see [BG92, Gar92, Gaf87, HS89, Per83]) are not directly applicable to an LVA. Furthermore, as argued before, a router using LVA must explicitly state which links it stops using.

Figure 2.1 helps to illustrate how LVA reduces storage and communication overhead compared to LSAs, even for the case of a fairly compact topology. An LSA would require

each router to maintain a copy of the entire topology, with an entry for each link in each direction. Because a router's source graph contains the links in all its preferred paths, a router using LVA has the same number of paths available as with an LSA for any type of routing that applies the same constraints at every router (e.g., shortest path routing, maximum capacity routing). In the case that the type of routing permits different constraints to be applied at different routers (e.g., policy-based routing), LVA offers only a subset of the paths available with complete topology information. However, such a subset of paths is the same as that obtained with any PVA, which are used in standard inter-domain routing protocols.

In the worst case, each router's source graph contains all the links in the network and the LVA requires the same communication and storage overhead as an LSA. The number of updates and size of updates in an LVA are bounded by a number proportional to the number and size of updates in an LSA, because in that case update messages contain *add* updates reporting changes to the parameters of network links, just as in an LSA.

2.3.2 Differences with PFAs

The basic idea in a PFA is for each router to maintain the shortest-path spanning tree reported by its neighbors (i.e., those routers connected to it through a direct link or a network), and to use this information, together with information regarding the cost of adjacent links, to generate its own shortest-path spanning tree. An update message exchanged among neighbors consist of a vector of entries that reports incremental or full updates to the sender's spanning tree; each update entry contains a destination identifier, a distance to the destination, and the second-to-last hop in the shortest path to the destination. Several PFAs have been proposed (e.g., see [CRKG89, Hum91, RF91]). Another PFA by Riddle [Rid84] is similar to the PFA method just mentioned in that a router communicates information regarding the second-to-last hop in the shortest path to each known destination. However, it uses exclusionary trees, rather than shortest-path spanning trees, and the cost of the link between the second-to-last hop and the destination, rather than the distance to

the destination. An exclusionary tree sent from router x to router y consists of router x's entire shortest-path tree, with the exception of the subtree portion that has node y as its root. Riddle's algorithm does not use incremental updates.

Of course, any path-finding algorithm can use the cost of the link between the second-to-last hop and the destination, rather than the distance to the destination. However, the set of preferred paths used by a node to reach other nodes need not constitute a tree in LVA and it is always a tree in a path-finding algorithm. There are many reasons why routers may want to communicate link-state information of preferred paths that do not correspond to a tree. For example, if multiple shortest paths are desired, a router will communicate links along multiple preferred paths to each destination. Because there can be multiple links leading to the same node in the subgraph of preferred paths communicated by a router, a router that receives an incremental update from a neighbor cannot simply assume that the link from node a to node b communicated by its neighbor can substitute any previously reported link from another node c to node b by the same neighbor; therefore, the update mechanisms used in path-finding algorithms to update the subset of link states maintained at each router are not applicable to LVA.

2.3.3 Differences with PVAs

With PVAs, routers exchange distance vectors whose entries specify complete path information for any destination they need to reach. The existing internet routing protocols based on PVAs (BGP [LR91] and IDRP [Rek93]) do not exchange link-state information per se, but such information can be exchanged in a PVA by including it as part of the information for each hop of the reported path in an update or update entry. This, however, would become very inefficient when the size of the network and the number of link-state parameters are large, or when multiple preferred paths to each destination are desired.

LVAs provide routers with all the path information that PVAs provide, but with far less overhead. This is the case because a router that uses a given link in one or more preferred

paths reports that link only once in LVA, while it has to include the link in each preferred path it reports using a PVA.

As we have noted, any routing algorithm that operates with less than a complete topology map reduces the set of all the possible paths that could be used to reach destinations when different constraints are applied at different nodes. In practice, this is not a problem, an LVA used for a given type of routing provides routers with exactly the same information than a PVA would for the same type of routing, because the routers' source graphs contain all the links in all the preferred paths selected by the routers. Therefore, any routing constraints or policies that can be supported with a PVA can be supported with an LVA, and practical routing protocols can be developed based on LVAs that provide the same functionality supported in BGP and IDRP, for example [Rek93].

A more subtle difference between LVA and a PVA using link-state information is that routers using LVA determine whether or not an update to a link state is valid based on the timeliness of that update alone, just as in an LSA. In contrast, a router using a PVA that communicates link-state information still has to operate on a path-oriented basis, i.e., the timeliness of an update refers to an entire path, not its constituent links; therefore, even if a router is able to ascertain that a given update is more recent than another, that update may still use link-state information that is outdated (e.g., regarding links that are far away in the path). To eliminate the possibility of using stale link-state information in an adopted path, each link of the path could be validated (with a sequence number, for example), but this becomes inefficient in a large internet.

2.4 Correctness of LVA

Theorem 1 below shows that LVA is correct for multiple types of routing under the assumptions introduced in Section 2.2 and the additional assumptions that there is a finite number of link cost changes up to time t_0, that no more changes occur after that time, and that routers can correctly determine which updates are more recent than others. Corollary 1 then shows that routing tables do not contain any permanent loops. Verifying that finite

sequence numbers and age fields can be used correctly to validate updates can be done in a manner similar to that used for finite sequence-numbering schemes used in LSAs [BG92].

Correctness for LVA means that, within a finite time after t_0, all routers obtain link-state information that allow them to compute loop-free paths that adhere to the constraints imposed by the local algorithms they use to compute preferred paths, and to forward packets incrementally.

Because our proof of correctness is intended for many different types of routing, not only shortest-path routing, we must specify what we mean by the correct operation of a path-selection algorithm. Consider the case in which each router in the network has complete and most recent topology information in its topology table and runs the same path-selection algorithm on it. In this case, it is evident that, for permanent loops to be avoided, the way in which the path-selection algorithm chooses routes must be deterministic. Assuming that the same deterministic path-selection algorithm is executed at each router using a complete and most recent copy of the topology, the preferred paths at any router for each destination constitute a directed acyclic graph (DAG). Furthermore, the union of the DAGs of any set of routers for the same destination in the network is also a DAG. Therefore, there are no permanent loops in the routing tables computed in this case.

Definition 2.1 *A correct path-selection algorithm is one that produces the same correct loop-free paths when it is provided with the same complete and correct topology information.*

This definition includes the requirements that ties for equally good paths must be broken according to the same rules. It is not necessary for all routers to use the same correct path selection algorithm to compute preferred paths for a given type of routing. All that is needed is for all the path-selection algorithms used for the same type of routing to produce the same loop-free paths when they are provided the same topology information.

Definition 2.2 *Two or more path-selection algorithms are compatible if they produce the same loop-free paths when they are provided with the same topology information.*

As we have stated in the description of LVA, all routers use the same path-selection algorithm, or compatible path-selection algorithms, to compute the same type of preferred paths (e.g., shortest path, maximum capacity), and report all the links used in all the preferred paths obtained through all the path-selection algorithms. Therefore, the rest of this section can assume that a single path-selection algorithm is executed at every router, and that every router runs the same path-selection algorithm.

Because the topology tables of different routers running LVA need not have the same information, we cannot use the notion of having all topology tables containing the same information to ensure correct paths. The following definition specifies what a topology table should have for loop-free paths to be produced in LVA.

Definition 2.3 *A router is said to have* consistent *link-state information in its topology table if it has the most recent link-state information regarding all the links for whom it is the head node, and the most recent link-state information corresponding to each of its neighbor's most recent source graph.*

While this definition of consistency defines a local state, describing a relationship between the content of the topology tables of neighboring nodes, it implies that if a node has information about a particular link in its table, then it has the most up-to-date information about that link.

Theorem 2.4 *A finite time after t_0, all routers have consistent link-state information in their topology tables and the preferred paths computed from those tables are correct.*

Proof: Because the deterministic path-selection algorithm used at each router is assumed to be correct, all the proof needs to show is that

1. All routers eventually stop updating their topology and routing tables, and stop sending update messages to their neighbors.

2. All routers obtain consistent link-state information needed to compute correct preferred paths within a finite time after t_0.

These two properties are proven in the following two lemmas.

Lemma 2.5 *LVA terminates within a finite amount of time after* t_0.

Proof: First note that there is a finite number of links in the network and that, by assumption, a finite number of link-state changes occur up to time t_0, after which no more changes occur. Also, by assumption, for each direction of a link whose parameters change, there is one router (the head node of the direction of the link) that must detect the change within a finite time; such a router updates its topology table and must then update its source graph. As a result of updating its source graph, the router can send at most one *add* update reporting the change in the state of the adjacent link, and at most one *add* or *delete* update for each of the links that have been added to or deleted from preferred paths as a result of the change in the adjacent link. Therefore, for any link l_i in the network, its head node can generate at most one update for that link after time t_0.

A given router x_1 that never terminates LVA must generate an infinite number of *add* or *delete* updates after time t_0. It follows from the previous paragraph that this is possible only if x_1 sends such updates as a result of processing update messages from its neighbors; furthermore, because the network is finite, x_1 must generate an infinite number of updates for at least one link l_1. Because the network is finite, at least one of those neighbors (call it x_2) must send to x_1 an infinite number of update messages containing an update for either link l_1 or some other link l_2 that makes x_1 generate an update for link l_1. It follows from the previous paragraph and the fact that the network is finite that x_2 can send an infinite number of updates regarding link l_1 or l_2 to x_1 only if at least one of its neighbors (call it x_3) generates an infinite number of updates for either link l_2 or some other link l_3 that makes x_2 generate updates regarding link l_1 or l_2. Because the network is finite, it is impossible to continue with the same line of argument, given that the head node of any link can generate at most one update for that link after time t_0. Therefore, LVA can produce only a finite number of updates and update messages for a finite number of link-state changes and must stop within a finite time after t_0. **q.e.d.**

Lemma 2.6 *All routers must have consistent link-state information in their topology databases within a finite time after* t_0.

Proof: The definition of consistent link-state information at a router implies that the router knows all the links it needs to compute correct preferred path, and that the router has the most recent link-state information regarding all the links in its topology table. Proving that the router receives all the link-state information required to compute correct preferred paths can be done by induction on the number of hops h of a preferred path. What needs to be shown is that the router knows all the links on that path within a finite time after t_0.

Consider some arbitrary preferred path from a router i to some destination. For $h = 1$, the preferred path consists of one of router i's outgoing links. Because of the basic assumption that some underlying protocol provides a router with correct information about its adjacent links within a finite time after the link-state information for such links changes, the lemma is true for this case. For $h > 1$, assume that the claim is true for any preferred path with fewer than h hops.

Consider an arbitrary preferred path of length $h > 1$ from some router i to a destination j. Let k be router i's successor on this path (i.e., the first intermediate router). Then, the subpath from k to j must have length $h - 1$, and it must be one of router k's preferred paths to j. Denote this path by P_{kj}. By the inductive hypothesis, router k knows all the links on P_{kj}. Because router i also knows (as in the base case) the most recent information about link l_k^i within a finite time after t_0, it suffices to show that router k indeed sends the link information in path P_{kj} to its neighbor router i.

Assume that P_{kj} is a new path for router k, then router k must update its source graph. Because P_{kj} is a new path for router k, the information in the updated source graph concerning P_{kj} is different than the information in the old source graph. Therefore, router k must include this information as *add* updates in the update message that it sends to its neighbors. Because router i is one of those neighbors, it must receive from k all the information on P_{kj} within a finite time after t_0.

By assumption router k can determine which link-state information is valid (i.e., up to date). Accordingly, if P_{kj} is already one of router k's preferred paths, but experiences a change in the information of some of its constituent links, then those links with updated link-state information will be considered different in the new source graph as compared to the old source graph. Therefore, router k must send the updated link-state information in P_{kj} to its neighbor i in *add* updates.

The same inductive argument holds for link-state changes resulting in links being deleted from a preferred path. In this case, an intermediate router that decides that a link should no longer be used in any of its preferred paths sends a *delete* update, which is propagated just like an *add* update. This completes the first part of the proof.

Having shown that a router receives the most recent information about the links used in its source graph within a finite time after t_0, it remains to be shown that it also receives the most recent information about all the links that are in its topology table, but not part of the source graph of preferred paths. There are two possible cases to consider of links in a router's topology table that are not used in its source graph: an adjacent link to the router, or a non-adjacent link is in the source graph reported by some of the router's neighbor. In the first case, it is obvious that the lemma is true because of the basic assumption of some underlying protocol providing the node with correct information about adjacent links within a finite time. The second case follows almost immediately from the first part of this proof. Because every neighbor of the router sends the appropriate *add* or *delete* updates about links added to or deleted from its source own graph, it must be shown that each such neighbor obtains consistent information about changes in its source graph, which was shown to be the case in the first part of this proof. **q.e.d.**

This concludes the proof of Theorem 2.4.

Corollary 2.7 *The routing tables created by LVA do not contain any permanent loop.*

Proof: Lemma 2.6 shows that the topology information at all routers is consistent within a finite amount of time after any change in link information. The topology information held at any router is a subset of the complete topology, and this subset contains all the information

needed at this router to compute the correct preferred paths. Therefore, the preferred paths computed from any router's subset of the topology information must be a subset of the DAG computed in the case of each router having complete topology information. Any subset of a DAG is still a DAG; and the union of any such DAGs also forms a DAG, because that union is also a subset of the DAG obtained with complete topology information. Hence, the routing tables computed by LVA with a correct path-selection algorithm do not contain permanent loops. **q.e.d.**

2.5 Complexity of LVA

This section quantifies the communication complexity (i.e., number of messages needed in the worst case), time complexity (number of steps), computation complexity, and storage complexity [Gar93] of LVA for shortest-path routing after a single link change.

2.5.1 Communication Complexity

The number of messages per link cost change is bounded by twice the number of links in the network. To prove that this is the case, it suffices to show that any update can travel each link at most twice. Assume that an update concerning link l arrives at some arbitrary node n for the first time; there are two possibilities to consider:

1. The link is used in the source graph of n. If this is the case, the corresponding link-state information is sent to some neighbor n_1 over some link l_1. There are two possibilities at this router:

   (a) n_1 uses l: If the information was already known and used at n_1, then no further update will be sent over l_1 (or any other link adjacent to n_1). If it was not previously known at n_1, then an update will be sent to all neighbors of n_1, including one over l_1 to n. From n, no further update with information concerning l will be sent over l_1, until newer information becomes available.

(b) n_1 does not use l: n_1 will not sent any update with information concerning l, in particular none over l_1.

2. The link is not used at n, in which case no further update will be sent.

From the above, it follows that the number of messages is at most in the order of the number of links in the network ($O(|E|)$).

2.5.2 Time Complexity

If the cost of links is not directly related to the delays incurred over such links, the number of steps required for any link change is $O(x)$, where x is the number of nodes affected by the change. This can be shown by the following argument: the information about a changed link travels along all the shortest paths that contained the link before the change, and also along all shortest paths that will contain the link after the change. No other router than those along the paths and their neighbors will be notified about the change.

In the worst case, all the affected routers lie along one long path, thus causing $O(x)$ communication steps. In general, the paths on which the information is forwarded together with the affected routers form a directed, acyclic graph, and the upper bound for the steps required is given by the length of the longest simple path in that graph.

Because link failures and recoveries are handled as special cases of link cost changes, and router failures are perceived by the network as link failures for all their links, it is clear that $O(x)$ communication steps are also incurred in these cases. The case of a recovering node involves the nodes getting the complete source graphs from its neighbors, which takes no more steps than the number of neighbors, before the links of the routers again are handled as changing their cost to some finite value. Hence, the same upper bound of $O(x)$ applies.

This worst case is the same as the complexity of the best DVA [Gar93]. On the other hand, if the link costs reflect the delay of the links, the complexity for LVA reduces to $O(d)$, where d is the (delay) diameter of the network. The reasons for this are that the information travels along the shortest paths and a router receiving new information can trust the neighbor that reports the most recent link-state for the associated link; most

importantly, the node will discard older information from other neighbors. Therefore, a router does not have to wait for link state updates to reach it through the slower paths, as is in the case in DVAs. The flooding technique used in LSAs also takes $O(d)$.

2.5.3 Complexity of Computations at Routers

The most important routines to analyze are *update* and *update_topology_table*. Most other procedures just call *update* with the appropriate input message. One part of *update* is the shortest path finding algorithm (Dijkstra) with a complexity of $O(|V|^2)$. The routing table can then be computed in time $O(|V|)$, and the update message can be assembled using *compare_source_graphs* in less that $O(|V|^2)$ time.

The complexity of the main loop of procedure *update_topology_table* is determined by the size of the update message. In the worst case, this message could contain information about every link, resulting in running time $O(|E|) \leq O(|V|^2)$. This case seems highly unlikely, though.

In "normal" cases, we would expect an update message to contain information about some path plus possibly a second path that has to be deleted. A path can have at most length $|V| - 1$, leading to an expected complexity of $O(|V|)$. The amount of work in the other loops is bounded by the number of nodes in the network.

All together, the overall worst case complexity for the procedure *update* is $O(|V|^2)$, mainly due to the shortest path algorithm. The corresponding complexity in a PVA is $O(|V|)$ for a single type of service.

Note that there also is the hidden complexity of accessing the topology table; this problem can be solved using a (dynamic) hash table, which has an expected constant access time.

2.5.4 Storage Complexity

In the worst case, the topology table of each router maintains the whole topology, making the storage requirement $O(|V|^2)$. In addition, both the shortest path tree and the routing

table require $O(|V|)$ storage, which is also the case for link state algorithms. Keeping track of reporting neighbors in LVA can be implemented by means of a bit vector. Because the identifiers of a router's neighbors can be stored in an ordered list, a single bit per neighbor suffices to indicate whether it is a member of the set of reporting nodes for a given link or not. Therefore, for each link in the topology table, the information about the reporting nodes adds only a constant amount of storage (which should be a few bytes for all practical purposes).

On the average, we expect the storage for the topology table to be by far smaller than $O(|V|^2)$. Because the goal is to keep as sparse a subset of the whole topology as possible (e.g., close to a tree), our conjecture is that the required storage space is closer to $O(|V|)$ for such simple types of routing as shortest-path routing. This seems realistic, even the small topology shown as example in Figure 2.1 revealed a significant saving of required space when compared to an algorithm that stores the complete topology at all routers. In contrast, the LSAs used today have to store the complete topology. Though the storage required for DVA is linear in the number of routers, routers have to store the routing tables of their neighbors. Therefore, DVAs' storage requirements really become $O(|V||N_k|)$ at router k, where N_k is the set of neighbors of node k.

2.6 Simulation

We compare LVA-SEN, DBF, and an ideal LSA in terms of the number of steps and updates that are required for the algorithm to converge (i.e., the algorithm stops sending messages), and the size of these updates. When a node receives an update message, it compares its local step counter with the sender's counter, takes the maximum and increments the count. In all three algorithms, update messages are processed one at a time in the order in which they arrive. Both LVA-SEN and LSA use Dijkstra's algorithm to compute the local shortest-path tree.

We performed two sets of simulation experiments: first, we used well known small topologies to analyze the behavior of LVA on a per event basis. Second, we used larger

topologies to analyze the scaling properties of LVA. For these simulations, we used both a well known topology as well as randomly generated topologies.

The simulation environment used for the first experiments was the OPNET modeler tool. OPNET facilitates the hierarchical modeling of communication networks. At the highest level, the topology and its parameters (such as node location and link bandwidth and delay) are defined. At the second level, the internal structure of the nodes is defined, specifying devices and protocols used for communication. Finally, at the lowest level, the protocols to be analyzed in OPNET are defined through finite state machines. The behavior in the states and in state transitions is defined by program fragments in the C programming language. The simulation kernel of OPNET is event driven.

For the second set of simulations, we implemented LVA and LSA in CPT (C++ Protocol Toolkit,) which was developed by Rooftop Communications. CPT is an object-oriented protocol framework which provides a class library for networking protocols that allows integrated development for simulations as well as embedded systems. Although intended for wireless systems, it also provides point-to-point facilities that we used in our implementations.

2.6.1 Small Topology

The results presented in this experiment are based on simulations for the DOE-ESNET topology [GZ94]; similar results were obtained for other smaller topologies. The graphs show the results for every single link changing cost from 1.0 to 2.0 (Fig. 2.4, 2.5, 2.6), every link failing (Fig. 2.7, 2.8, 2.9) and recovering (Fig. 2.10, 2.11, 2.12), as well as every node failing (Fig. 2.13, 2.14, 2.15) and recovering again (Fig. 2.16, 2.17, 2.18). All changes were performed one at a time, and the algorithms had time to converge before the next change occurred. The ordinate of Figures 2.4 to 2.12 and Figures 2.13 to 2.18 represent identifiers of the links and the nodes, respectively, that are altered in the simulation. In Figures 2.4, 2.7, 2.10, 2.13, and 2.16, the data points show the number of update messages sent, in

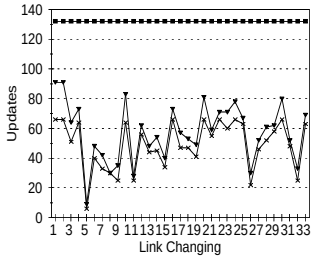

FIGURE 2.4: Updates for link changes

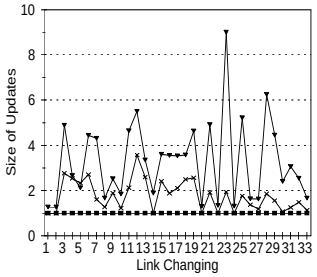

FIGURE 2.5: Size of updates for link changes

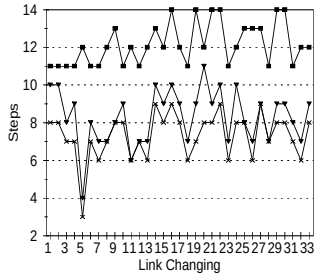

FIGURE 2.6: Steps for link changes

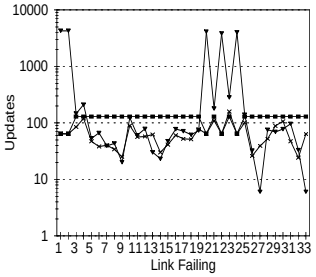

FIGURE 2.7: Updates for link failures

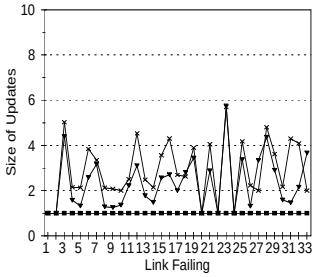

FIGURE 2.8: Size of updates for link failures

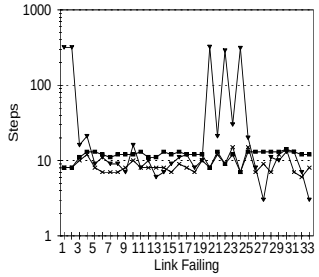

FIGURE 2.9: Steps for link failures

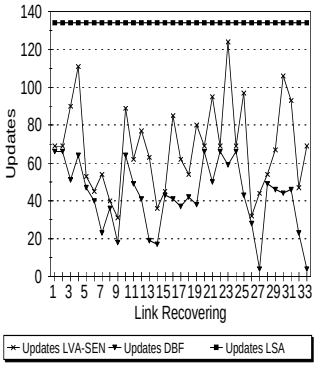

FIGURE 2.10: Updates for link recoveries

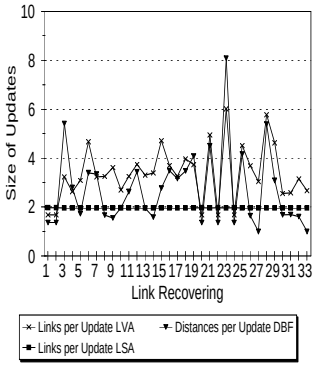

FIGURE 2.11: Size of updates for link recoveries

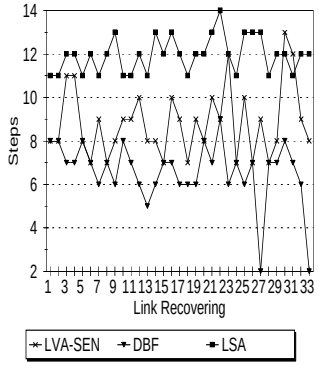

FIGURE 2.12: Steps for link recoveries

Figures 2.5, 2.8, 2.11, 2.14, and 2.17 they show the size of these updates, and in Figures 2.6, 2.9, 2.12, 2.15, and 2.18, they show the number of steps needed for convergence.

LSA shows almost constant behavior for all single link cost changes (Figures 2.4, 2.6) because the same link-state update must be sent to all routers. In contrast, DBF and LVA-SEN propagate updates to only those routers affected by the link-cost change; LVA-SEN is

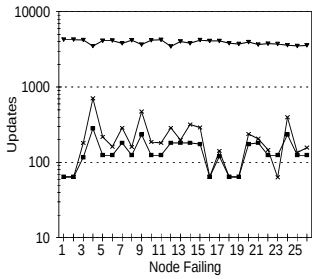

FIGURE 2.13: Updates for node failures

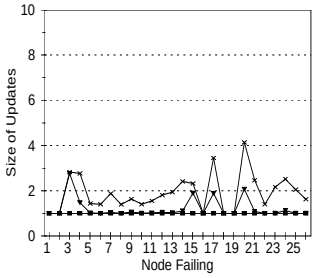

FIGURE 2.14: Size of updates for node failures

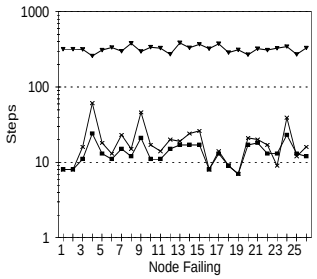

FIGURE 2.15: Steps for node failures

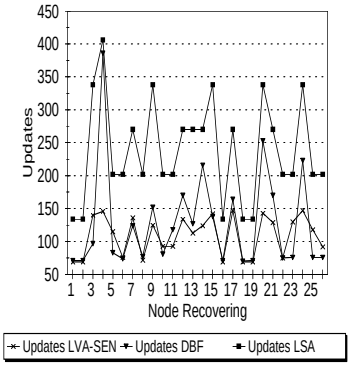

FIGURE 2.16: Updates for node recoveries

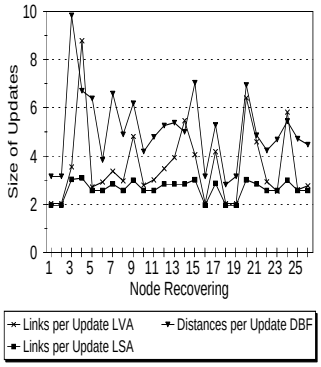

FIGURE 2.17: Size of updates for node recoveries

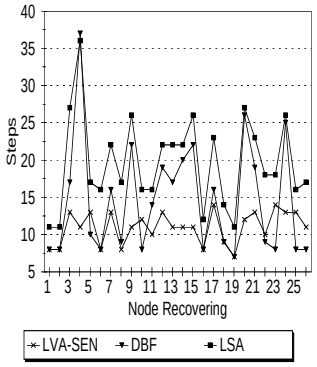

FIGURE 2.18: Steps for node recoveries

the most efficient of the three algorithms. Each update message contains one link state in LSA, and an average of 1.77 links in LVA-SEN. Figures 2.7 and 2.9 show similar behavior of the three algorithms for link failures, the exception being DBF suffering from 'counting to infinity' in some cases. In almost all cases, LVA-SEN needs fewer update messages and fewer steps than LSA; the average size of an LVA-SEN messages is 2.89 links.

When a failed link recovers, DBF is superior to both LVA-SEN and LSA. LSA exhibits the same behavior as with link-cost changes. With the exception of link 30, LVA-SEN is always better than LSA (Figures 2.10 and 2.12). The average LVA-SEN message is slightly more than three links; and LVA-SEN almost always requires less information to be sent than LSA and DBF, because messages in LSA are no longer one-link long due to the packets containing complete topology information sent over the recovering link.

For failing nodes, LSA usually has the best performance of the three algorithms. DBF always suffers from 'counting to infinity'. In almost all cases, LSA needs fewer steps and updates (Figures 2.13 and 2.15) than the other algorithms. The average message size for LSA is one link and 1.9 links for LVA-SEN.

DBF is superior to LSA when a node recovers, and LVA-SEN performs even better than DBF. LVA-SEN needs fewer steps and updates than the other algorithms (Figures 2.16 and 2.18). The average message size for LVA-SEN is 3.60 links), but of the three algorithms it still requires the least amount of information to be sent through the network.

Overall, the results of our simulations are quite encouraging. In terms of its overhead, LVA-SEN behaves much like DBF when link costs change and is always faster and produces less overhead traffic than LSA when resources are added to the network, and behaves much like the ideal LSA when links or routers fail. This is precisely the desired result, and indicates that LVAs are desirable even if multiple constraints are not an issue. It is apparent that LSA performs better than LVA only after node failures. The reason for this is that a failed node always impacts one preferred path for each node (i.e., every node's path to the failed node), which implies at least one *delete* update, and may also impact additional preferred paths of a subset of the nodes for other destinations, which may imply various *add* and *delete* updates for different links. Therefore, while LSA has to report only what happened to the links adjacent to the failed node, LVA needs to also add other links to bypass the failed node.

A simple way to improve the performance of an LVA after a node failure is the following: when a router detects a link failure or receives a *delete* update from a neighbor reporting a link failure, the router waits for a short hold-down time proportional to one propagation time between neighbor routers to receive updates from other neighbors (which may contain *delete* updates for links adjacent to the same destination in the case of a node failure) before it updates its source graph and generates its own updates. Although the time a router takes to propagate updates due to link failures increases, the associated control traffic decreases; furthermore, routers away from the failed resource are more likely to get all the *delete*

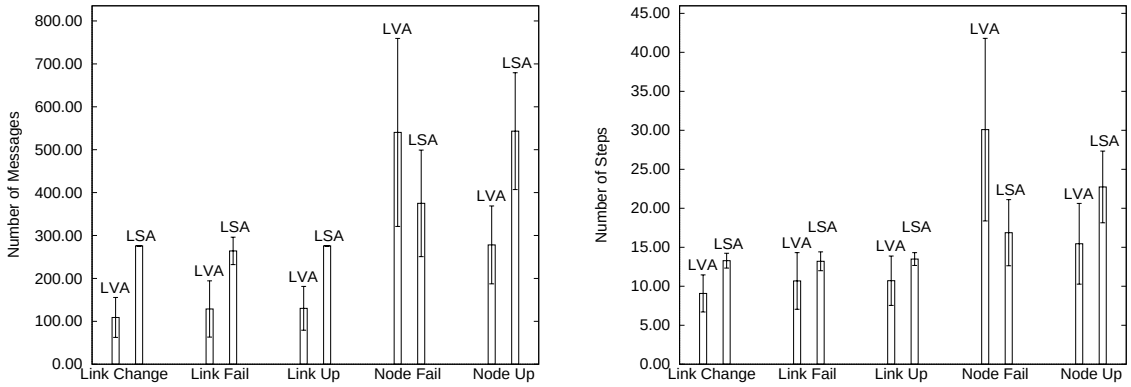

FIGURE 2.19: Messages and Steps for ARPANET topology

updates associated with a node failure in the same update message from any one of its neighbors, which means that the router will identify a destination that has failed and will not try to add links that no longer exist to its source graph to try to reach or use such destination as part of its preferred paths.

2.6.2 Larger Networks

Similarly to the results described in section 2.6.1, we performed changes (i.e., changes in link cost, link failures, link recoveries, node failures, and node recoveries) one at a time. We compare LVA and LSA in terms of the number of update messages and steps needed for convergence, and the average number of links in the topology table.

The figures in the following subsections show the results of these simulations. For all event types, we show the average number of messages sent and steps take, together with the standard deviations. Since all results are shown for specific topologies and all possible single events are covered in the simulations, no sampling errors can be given.

ARPANET Topology

The well known topology that we chose for our simulations is the ARPANET topology[GZ94]. This topology has 47 nodes and 69 links, giving it an average degree of 2.94.

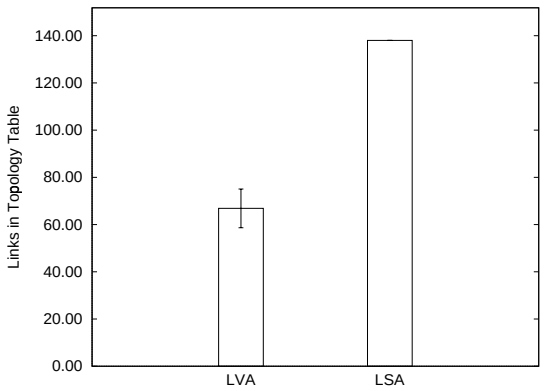

FIGURE 2.20: Size of Topology Tables for ARPANET topology

Figure 2.19 shows the results for this topology. In all cases of single resource changes (link changes, failures, and recoveries,) LVA sends on the average about half as many packets as LSA, and it needs fewer steps to converge.. As in the simulations presented in the previous section, LSA outperforms LVA when nodes fail. However, when nodes are added to the network, LVA is significantly better than LSA, and taken together, a node that fails and comes back up later causes less traffic in LVA than in LSA.

Figure 2.20 shows the average table sizes for both LVA and LSA. Obviously, the tables at all nodes have the same size when LSA is used, because the complete topology is replicated everywhere. On the average, topology tables produced by LVA are about half as large as those produced by LSA. Clearly, this is the cause for the reduced communication overhead. On the other hand, when nodes fail, additional links must be added to many tables (and others must be deleted,) causing more updates to be generated for LVA when compared to LSA.

Overall, the simulations using the ARPANET topology confirmed our findings from the small topologies. LVA outperformed LSA in terms of messages sent and number of steps until convergence, and its storage requirements are significantly lower than those of LSA.

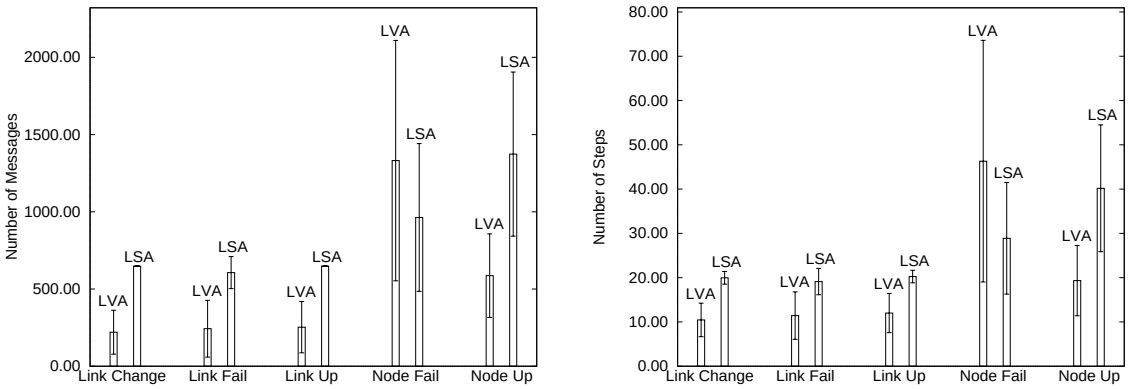

FIGURE 2.21: Topology a

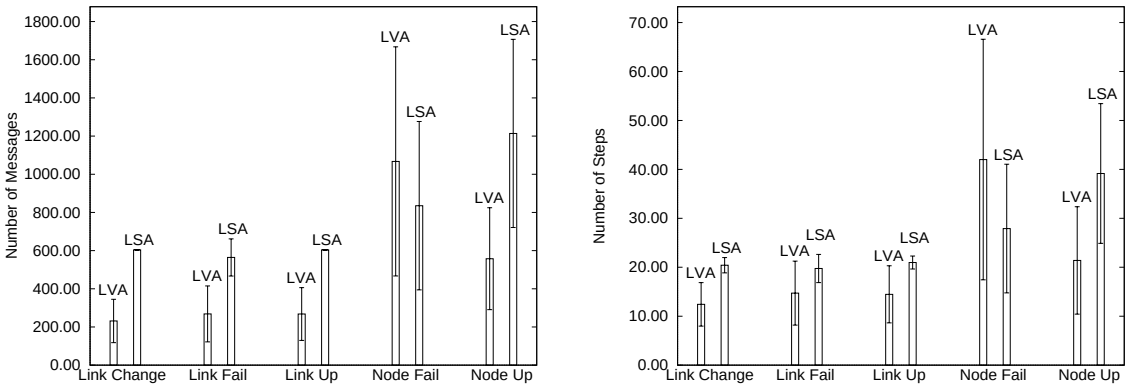

FIGURE 2.22: Topology b

2.6.3 Random Topologies

Random topologies were obtained using the exponential model proposed in [ZCB96]. All topologies generated this way have 100 nodes (the maximum number allowed in CPT) and a varying number of links. The topologies used are flat versions of the topologies that we used for the simulation of hierarchical routing algorithms, a more detailed description of how we obtained them can be found in chapter 4.

Figure 2.21 shows the results for a topology with 100 nodes and 162 links, giving it an average degree of 3.24. The maximum degree of a node in the network is 9. Figure 2.22 shows results for another topology with 100 nodes. This topology has an average degree of

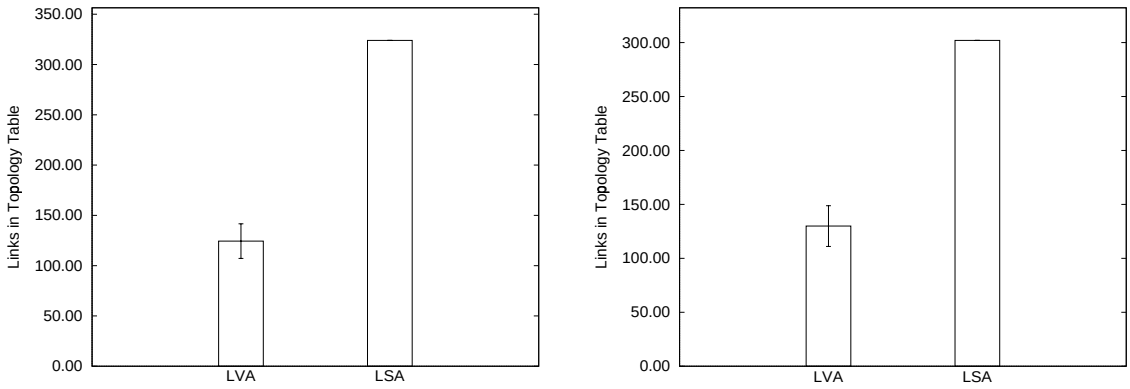

FIGURE 2.23: Size of Topology Table for Topologies a and b

3.02 (151 links) and a maximum degree of 9. These two topologies are representatives of two classes of topologies as described in chapter 4.

The results from these random topologies confirm the results obtained using well known topologies. In the topologies shown (and in other topologies obtained using the same methods,) LVA clearly outperforms LSA in all events except for link failures.

Figure 2.23 shows the average sizes of the topology tables stored at nodes in these topologies. Topology tables used in LVA are on the average less than half as big as the ones used in LSA. In terms of storage overhead, LVA outperforms LSA by an even wider margin than in the smaller topologies that we used for simulations.

2.7 Summary

We have presented a new method for distributed routing in computer networks and internets using link-state information. LVAs enjoy nice scaling properties: like DVAs, LVAs scale well with the number of destinations by aggregating information; like LSAs, LVAs scale well with the number of service types because routers communicate link properties, not path properties in their updates. An important contribution of this chapter is to show that LVA is correct under different types of routing, assuming that a correct mechanism is used for routers to ascertain which updates are recent or outdated.

LVAs open up a large number of interesting possibilities for routing protocols of large internets. LVAs can be used to develop intra-domain routing protocols that are based on link-state information but require no backbones or areas, and can take advantage of simple aggregation schemes developed for DVAs. LVA can also be applied to inter-domain routing protocols that provide the same functionality of BGP/IDRP while reducing the overhead incurred in communicating and storing updates.

2.8 Appendix: Pseudo Code for LVA-SEN

LVA-SEN Variables and Data Structures at Node i

(j,k)	link from j to k
TT_i	topology table
(j,k,l,sn,r)	entry in TT_i for (j,k)
l	cost of the link
sn	sequence number of link
r	set of reporting nodes for link
ST_i	source graph at router i
t	sequence number of router processing event
N_i	set of neighbors of router i
$(j,k,l,sn,type)$	entry in update message
$type$	add or delete operation

Link Up

This Procedure is called when the underlying protocol detects connectivity to a new neighbor j. The nose j is added to the set of neighbors. Then, an update containing cost of new link is created and procedures update is called to process the update. In addition, and update message containing the complete source graph is sent to the new neighbor.

procedure link_up(i,j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of destination of link
  begin
    $N_i = N_i \cup \{j\}$
    update $(i,i,\{(i,j,l^i_j,t,\text{add})\})$
    u_message = $\emptyset$
    for all $(k,m) \in ST_i$ **do**
      u_message = u_message $\cup$ ($TT_i(k,m)$, add)
    end for
    send(j, u_message)
  end link_up

Link Down

This procedure is called when the underlying protocol indicates loss of connectivity to neighbor j. First, j is removed from set of neighbors. Then, j is removed from all sets of reporting nodes, a link is deleted if j was the only reporting node. An update containing the infinity cost for the link is created and procedure update called to process it.

procedure link_down(i, j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of destination of link
begin
message $= \emptyset$
$N_i = N_i - \{i\}$ (but keep sequence number)
for all $(k, m) \in TT_i$ **do**
$TT_i(k, m).r = TT_i(k, m).r - \{j\}$
if $TT_i(k, m).r = \emptyset$ **or** $TT_i(k, m).r = \{i\}$ **then**
message = message $\cup$ $\{(TT_i(k, m), \text{delete})\}$
end if
end for
message = message $\cup$ $\{(i, j, \infty, t, \text{delete})\}$)
update (i, i, message)
end link_down

Link Change

This Procedure is called when the underlying protocol detects a change in the cost of a link. An update containing the new cost of the link is created and procedure update is called to process this update.

procedure link_change(i)
– – Parameter:
– – i: name of node that executes procedure
begin
update$(i, i, \{(i, j, l_j^i, t, \text{add})\})$
end link_change

Node Up

This procedure is called when a node comes up. The node starts with an empty topology table, source graph, and routing table. Links to all neighbors are added to the topology table. Then, all neighbors are queried for their complete source graph, and procedure update is called to process the answers. After receiving answers from all neighbors, the new source graph and routing table are built and an update is sent to the neighbors.

```
procedure node_up (i);
- - Parameter:
- - i: name of node that executes procedure
    begin
        t = 0
        TT_i = ∅
        ST_i = ∅
        message = ∅
        N_i = {x | ∃(i, x), l_x^i < ∞}
        for all x ∈ N_i do
            TT_i = TT_i ∪ (i, x, l_x^i, t, {i})
            ST_i = ST_i ∪ (i, x, t)
        end for
        build new routing table
        for all x ∈ N_i do
            send (x, query)
        end for
        answers_received = 0
        while answers_received < |N_i| do
            receive (answer)
            t = max{t, answer.t}
            update_topology_table (i, answer, u_message)
        end while
        t = t + 1
        for all x ∈ N_i do
            TT_i(i, x).t = t
        end for
        build_shortest_path_tree(i, TT_i, NewST_i)
        build new routing table
        ST_i = ∅
        u_message = ∅
        compare_source_graphs (ST_i, NewST_i, u_message)
        for all x ∈ N_i do
            send (x, u_message)
        end for
        ST_i = NewST_i
        t = t + 1
    end node_up
```

Answer Query

This procedure is called when a topology query is received from a neighbor. An update is sent containing the complete source graph.

```
procedure answer_query(i, j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of neighbor that sent query
    begin
        if (i, j) ∉ ST_i then
            ST_i = ST_i ∪ (i, j)
            build routing table
        end if
        for all (k, m) ∈ ST_i
            u_message = u_message ∪ TT_i(k, m)
        end for
        u_message.t = t_j
        send (j, u_message)
    end answer_query
```

Update

This procedure is called when an update message needs to be processed. It processes all updates in the message. If the content of the message caused change in the topology table, a new source graph and routing table are computed and an update message is sent to neighbors.

```
procedure update (i, n, message)
– – Parameters:
– – i: name of node that executes procedure
– – n: name of neighbor that sent update message
– – message: update message to be processed
    begin
        u_message = ∅
        updated = update_topology_table (i, message, u_message)
        if u_message ≠ ∅
            send (n, u_message)
        end if
        u_message = ∅
        if updated then
            build_shortest_path_tree (i, TT_i, NewST_i)
            build routing table
            compare_source_graphs (i, ST_i, NewST_i, u_message)
            remove marked links from TT_i
            if u_message ≠ ∅ then
                for all x ∈ N_i do
                    send (x, u_message)
                end for
            end if
            ST_i = NewST_i
            t = t + 1
        end if
    end update
```

Update Topology Table

The procedure update_topology_table is called to process an update message. All link-state updates contained in the message are processed using procedures process_add_update and process_delete_update to update the topology table accordingly.

```
procedure update_topology_table (i, message, u_message)
– – Parameters:
– – i: name of node that executes procedure
– – message: update message to be processed
– – u_message: new update message
    begin
        updated = false
        for all m = (j, k, l, sn, type) do
            if type = add then
                updated = process_add_update (i, message.source, j, k, l, sn)
            else
                updated = process_delete_update (i, message.source, j, k, l, sn)
            end if
            if j = message.source and j ∈ N_i then
                store sequence number of neighbor
        end for
        return updated
    end update_topology_table
```

Process ADD Update

This procedure processes an ADD update. If it is an up-to-date update, the link is added to the topology table if it was not listed before; the link-state information is updated and the sender of the update is added to the set of reporting nodes if it is already in the table. If the update is outdated, a message containing the correct information is generated to be sent to the neighbor that sent the update.

```
procedure process_add_update (i, n, j, k, l, sn , u_message)
– – Parameters:
– – i: name of node that executes procedure
– – n: sender of update
– – j, k: head and destination of link
– – l: cost of link
– – sn: sequence number of update
– – u_message: new update message
    begin
        if (j, k) ∈ TT_i then
            if TT_i(j, k).sn < sn then
                TT_i(j, k) = m
                TT_i(j, k).r = {n}
                updated = true
            else if TT_i(j, k).sn = sn
                        and i ≠ n then
                TT_i(j, k).r = TT_i(j, k).r ∪ {n}
                updated = true
            end if
        else if (i ≠ j or n = i)
                    and (TT_i(j, k).sn <= sn) then
            TT_i = TT_i ∪ m
            updated = true
        end if
        if TT_i(j, k).sn > sn then
            – – generate update to send up-to-date
            – – information to neighbor
            if (j, k) ∈ ST_i then
                u_message = u_message ∪ (TT_i(j, k), add)
            else
                u_message = u_message ∪ (TT_i(j, k), delete)
            end if
        end if
    end process_add_update
```

Process DELETE Update

This Procedure processes a DELETE update. If the update is up-to-date, the sender of the update is removed from the set of reporting nodes. If it was the only reporting node, the link is deleted. If the update is outdated, a message containing the correct information is generated to be sent to the neighbor that sent the update.

```
procedure process_delete_update (i, n, j, k, l, sn , u_message)
– – Parameters:
– – i: name of node that executes procedure
– – n: sender of update
– – j, k: head and destination of link
– – l: cost of link
– – sn: sequence number of update
– – u_message: new update message
  begin
    if TT_i(j,k).sn < sn then
      if (j,k) ∈ TT_i then
        mark (j,k) as deleted
        updated = true
      else
        TT_i(j,k).sn = sn
      end if
    else if TT_i(j,k).sn = sn then
      if (j,k) ∈ TT_i then
        TT_i(j,k).r = TT_i(j,k).r − {n}
        if (TT_i(j,k).r = ∅ or TT_i(j,k).r = {i})
              and i ≠ n then
          mark (j,k) as deleted
          updated = true
        end if
      end if
    else if TT_i(j,k).sn > sn then
      if (j,k) ∈ ST_i then
        – – generate update to send up-to-date
        – – information to neighbor
        u_message = u_message ∪ (TT_i(j,k), add)
      else
        u_message = u_message ∪ (TT_i(j,k), delete)
      end if
    end if
    if TT_i(j,k).l = ∞ and TT_i(j,k).sn < sn then
      TT_i(j,k).sn = sn
    end if
  end process_delete_update
```

Compare Source Graphs

This procedure is used to produce an update to be sent to all neighbors. It compares the old and the new spanning tree and generates an update message containing the differing link states.

```
procedure compare_source_graphs (i, ST_i, NewST_i, u_message)
– – Parameters:
– – i: name of node that executes procedure
– – ST_i: old spanning tree at node i
– – NewST_i: new spanning tree at node i
– – u_message: new update message to be generated
   begin
      – – first, generate add updates for all links that are in
      – – the new spanning tree but were not in the old one.
      for all (j,k) ∈ NewST_i, ((j,k) ∉ ST_i
            or NewST_i(j,k).sn > ST_i(j,k).sn) do
         u_message = u_message
                        ∪ (j, k, TT_i(j,k).sn, TT_i(j,k).l, add)
      end for
      – – now, generate delete updates for links that were in
      – – the old spanning tree but are not in the new one.
      for all (j,k) ∈ ST_i, (j,k) ∉ NewST_i do
         if i = j then
            u_message = u_message ∪ (j, k, t, TT_i(j,k).l, delete)
         else
            u_message = u_message
                        ∪ (j, k, TT_i(j,k).sn, TT_i(j,k).l, delete)
         end if
      end for
   end compare_source_graphs
```

Chapter 3

Update Verification

3.1 Introduction

Disseminating link-state (topology) information reliably is essential to many internet routing protocols proposed or implemented to date. This dissemination can take the form of broadcast, in which every network node (router) maintains the same topology map [BG92], or selective distribution, in which each node maintains only the subset of the topology map it needs to perform correct routing, as described in the previous chapter. Of course, in a very large internet, network resources (links and nodes) must be aggregated into clusters or areas to reduce the amount of information each node needs to store and process; however, because the focus of this chapter is on the basic algorithm used for disseminating topology information in a network a flat network organization is assumed for ease of presentation.

Broadcast of link states can be accomplished by flooding or building a spanning tree over which link states are distributed [HS89, Gar92]. This chapter focuses on flooding because of its simplicity and popularity. Examples of standard internet routing protocols based on the flooding of link states are OSPF [Moy94], IS-IS [ISO89] and NLSP [Nov94]. In addition, the inter-domain policy routing (IDPR) architecture [EST93] and the Nimrod architecture for scalable internet routing [CCS95] are both based on flooding. These protocols and architectures use the same basic approach for the flooding of topology information, which

we simply call *intelligent flooding protocol* or IFP in the rest of this chapter (e.g., see [Per83, PVL92]).

According to IFP, each network router ascertains the state of its neighboring links and reports this in what we will call a *link-state update* (LSU); for simplicity, we assume that an LSU reports the state of only one link adjacent to a given source. The basic problem then becomes one of broadcasting the most recent LSUs of each source to every router in the network. Once this is accomplished, each router has a topology map from which it can compute the desired paths to destinations. To flood LSUs, IFP uses sequence numbers to validate the most recent LSU from each source; a router receiving an LSU accepts the LSU as valid (i.e., recent) only if the received LSU has a higher sequence number than the sequence number of the LSU stored from the same source.

Because the sequence-number space available in a routing protocol is finite, IFP must operate with finite sequence numbers. To accomplish this, a linear sequence-number space is used together with an age field, and large enough that the maximum sequence number should be reached only in very rare circumstances. Each LSU specifies a sequence number and an age. The source starts its first LSU with a sequence number equal to 0, and sends a new LSU with a higher sequence number after either detecting a change in the state of an adjacent link, or after reaching a maximum time with no state changes in adjacent links. Each LSU sent by the source specifies the current sequence number and the maximum age for that LSU (in the order of an hour in today's protocols). No more LSUs from the same source are accepted when the sequence number reaches its maximum value, until the LSU is erased due to aging. Aging means that every router that accepts an LSU decrements its age by at least one and also decrements the age while the LSU sits in memory. A router rebroadcasts an LSU when it reaches age 0. If a router receives a valid LSU with an age of 0, the router ignores the LSU if it does not have a stored LSU from the same source, and rebroadcasts the LSU if it has a copy of the LSU; this ensures that nodes age LSUs at a similar pace. LSUs must be sent reliably between neighbors.

The *link-vector algorithm* (LVA) introduced in [GB95] is based on the selective distribution of topology information, rather than on flooding. The purpose of this algorithm is to allow a router to maintain only the link-state information it needs to reach a destination, rather than the entire topology map. Each router maintains a subset of the topology map corresponding to its adjacent links and the links that its neighbor routers have reported as being used in their paths to destinations. The router uses this information to compute its own paths to destinations, and reports to its neighbors the states of only those links used in the chosen paths. In addition, the router tells its neighbors which links it no longer uses to reach destinations. A basic assumption for the correct operation of the LVA implementation introduced in [GB95] is that routers can determine whether an update contains up-to-date information using the same update validation scheme used for LSUs in IFP.

The inherent limitation with the above method is that the age field must be very long to avoid situations in which, due to aging, routers loose LSUs that are still valid. Furthermore, because every LSU must expire in a finite time, the source of each LSU must retransmit new incarnations periodically in the absence of link-state changes. In practice, aging of sequence numbers introduces additional communication overhead. Furthermore, after resource failures that isolate any portion of the network from a source of LSUs, old LSU information can be erased only after reaching its maximum age.

We propose a new algorithm that achieves fast dissemination of up-to-date link-state information without periodic updates or age fields. The algorithm is based on a finite and linear sequence-number space and diffusing computations [DS80]. Most prior applications of diffusing computations to routing [JM82, Gar93] have focused on the dissemination of information regarding distances to destinations. The application of diffusing computations to the dissemination of link-state information has been proposed before only in the context of building a spanning tree over which link-state updates are distributed [Gar92]. Our update algorithm can be applied to standard internet routing protocols based on flooding, eliminates the need for periodic flooding of LSUs, can dramatically reduce the amount of time in which obsolete LSUs can be erased after resource failures or new LSUs can be copied

throughout the network after resource recoveries, and incurs limited overhead. The latency of our algorithm in reseting sequence numbers and erasing old information is bounded only by the time it takes for an LSU to traverse the network, rather than by a global timer, which is the case in all previous reset schemes used or proposed to date (e.g., see [PVL92, APV94]). We note that, just as with other link state algorithms, the routing of user messages proceeds while the dissemination of link-state information is taking place.

The following section states the goals for our reset mechanism. Section 3.2 describes the new reset algorithm in the context of flooding as well as selective diffusing of topology information. Section 3.3 verifies that the new reset algorithm works correctly within the context of selective dissemination of topology information, and that the resulting routing protocol ensures that all routers receive the information they need to make correct routing decisions. We chose to address correctness in the context of selective dissemination because, as we will show, selective dissemination represents a generalization of flooding. Section 3.5 analyzes the complexity of the selective dissemination algorithm. Section 3.6 summarizes the applicability of our results.

3.2 Objectives of The Reset Algorithm

The objective of the reset algorithm is threefold:

- When the sequence number of an LSU wraps around at its source (i.e., the sequence-number space is exhausted and the sequence number is reset to 0), all the routers affected by the LSU are forced to synchronize with the source in such a way that, in the absence of topology changes, all other sequence numbers for the same LSU are purged and all routers affected by the LSU reset its sequence number to 0, before the source can increment the sequence number.

- After a malfunction that makes a router reset the sequence number of an LSU for which it is not the source, the router forces either the source or another router to provide the correct sequence number.

- After a resource failure, routers with no physical path to the source of an LSU erase the LSU within a finite time proportional to the time it takes to traverse their connected components.

Dijkstra and Scholten's basic algorithm was used in [Gar89] to provide loop freedom in topology broadcast algorithms, assuming that the intelligent flooding protocol was used with no changes (i.e. aging and periodic retransmission are used to handle finite sequence numbers).

A reset algorithm for IFP with goals similar to the above three has been sketched in [APV94]. According to this algorithm, whenever the sequence number of an LSU reaches its upper bound at some router, this router makes a reset request. When a request reaches a router other than the source, that router resets its sequence number to 0 and forwards the request. When the source of the LSU receives the request, it sets its sequence number to 1 and broadcasts its most recent LSU. This type of reset has two problems: erroneous LSU information has to propagate all the way to the source before it can be erased (a technique first suggested by Humblet [BG92]), and other than having a global timer for garbage collection, there is no provision for erasing an obsolete LSU after the failure of the source of the LSU or the partition of the network. The following section outlines how our reset algorithm supports the three goals stated above.

3.3 Description of Reset Algorithm

To validate LSUs with no need for periodic transmissions or age fields, we present a sequence-number reset algorithm based on *diffusing computations* [DS80]. The reset algorithm can be applied to both the replication of the same LSU at every router, or the selective dissemination of LSUs. Replicating LSUs at every router is a special case of selective dissemination of LSUs, and there are only two important simplifications: The first is that, because every router must receive every LSU, there is no need for a router to request its neighbors to delete any LSU. The other simplification is that a router does not have to decide which LSU to

propagate depending on the link constituency of its paths to destinations; the router simply propagates each valid LSU.

We describe our reset algorithm within the context of both selective dissemination and flooding of link states. A network is modeled as an undirected graph $G = (V, E)$, where V is the set of nodes (routers) and E is the set of edges (links). Each link exists it both directions at any time, and there is a (possibly different) cost assigned in each direction. An underlying protocol assures that every node detects changes in link states within a finite amount of time. All such changes are processed one at a time, and in the order in which they are detected.

Assume that a protocol is used for the dissemination of link-state information and the maintenance of topology and routing tables. This protocol, be it based on flooding or selective dissemination of link states, must use certain message formats to exchange link states among adjacent routers. We have called such messages LSUs; we assume that an LSU specifies who originates it, a sequence number, the state of the link, and an add or delete instruction in the case of selective dissemination. Sequence numbers are assumed to be drawn from a finite and linear sequence-number space. In the same way that some routing protocols based on routing do (e.g., OSPF), we assume that LSUs are exchanged reliably between neighbors. When a router sends an LSU in a message, it waits for acknowledgments from all its neighbors, and retransmits the message with the LSU to a neighbor if it does not receive an ack after a timeout. Connectivity with a neighbor is assumed lost after a number of unsuccessful message transmissions.

In steady state, the topology of the network is stable and none of the sequence numbers wraps around. In such a case, nodes execute the flooding or selective dissemination protocol by exchanging LSUs as summarized in Section 3.1 for IFP and LVA. In this case, a node that originates an LSU and sends it to its neighbors only needs to receive acknowledgments from them stating that they have received the LSU. Node i does not need to know that either all of the network nodes have received a copy of the LSU (in the case of flooding), or

that all the nodes using the link reported in the LSU have received the LSU (in the case of selective dissemination).

There are three cases in which a node must ensure that all the nodes that need to know about the state of a given link receive the new information for the link and adapt the correct sequence number. These cases are the following:

- The node needs to reset the sequence number for one of its outgoing links.
- The node detects failure one of its links.
- The node detects that it has no physical path to the head of a remote link.

This is accomplished by means of two additional types of update message entries: *queries* and *replies*. Both are reliably transmitted between neighbors by means of message acknowledgment and retransmissions, as described for the case of LSUs. Queries have the same fields of an LSU. Replies do not need to transmit link information, but may carry a tag signaling the possibility of an error. Based on these queries and replies, our reset algorithm operates in a manner very similar to Dijkstra and Scholten's algorithm for diffusing computations [DS80].

A node that needs to reliably distribute information about a link through the network and detect the termination of this, sends queries instead of LSUs to all its neighbors and then waits until it receives a reply from each neighbor. An acknowledgment signals that a neighbor has received an LSU (or a query) correctly. A reply signals that a neighbor and all nodes connected through that neighbor that need to process the query have done so. A node is said to be in *active* mode (or state) when it is waiting for replies, otherwise it is *passive*. A passive node receiving a query for a given link follows the same pattern, it forwards the query to all its neighbors, waits for their replies, and, upon reception of the last reply, sends a reply to its predecessor in the diffusing computation, i.e., to the node from which it received the query that caused its transition to active state. If an active node receives another query, then it simply sends a reply back to the neighbor that sent the query.

An update message may contain queries and replies, as well as plain LSUs from the underlying routing protocol. When an update message is received, the node first processes all the replies, then the LSUs, and at last the queries that are included. Then, it assembles the packets to be sent to its neighbors. The replies must be processed first so that updates can be queued if the respective reply is in the same packet. Processing updates before queries may speed up convergence of the diffusing computations, because more complete up to date topology information is available to determine the correct action for the query.

3.3.1 Reset for Flooding

In the case of flooding, the complete topology information needs to be replicated at every node. The pseudo code in the appendix of this chapter (section 3.7) formally specifies a flooding protocol based on our reset algorithm. For simplicity, all messages, which can contain LSUs, queries, and replies, are assumed to be transmitted correctly over an operational link.

A passive node processes LSUs according to the rules for intelligent flooding. If an active node receives an LSU, it must check whether it already received a reply from the sender of the LSU. If this is the case, then the update must be buffered because it contains more recent information than the query did. There must be a separate buffer for each neighbor, but only the latest LSU must be kept. In addition, the buffer is flushed when a query is received subsequently over the same link.

When a passive node receives a reply, this reply is simply discarded. An active node receiving a reply checks if this is the last reply that it expects; if this is the case, the node goes into passive state and sends a reply to its predecessor in the diffusing computation (unless it is the source of the diffusing computation, in which case the diffusing computation is terminated). It then processes buffered LSUs. In the case that the state of the link changed since the node became active (i.e., the buffered LSUs contained more recent information), LSUs are sent to all neighbors. If the reply carried an error tag, all subsequent replies for this diffusing computation that the node sends also carry such a tag.

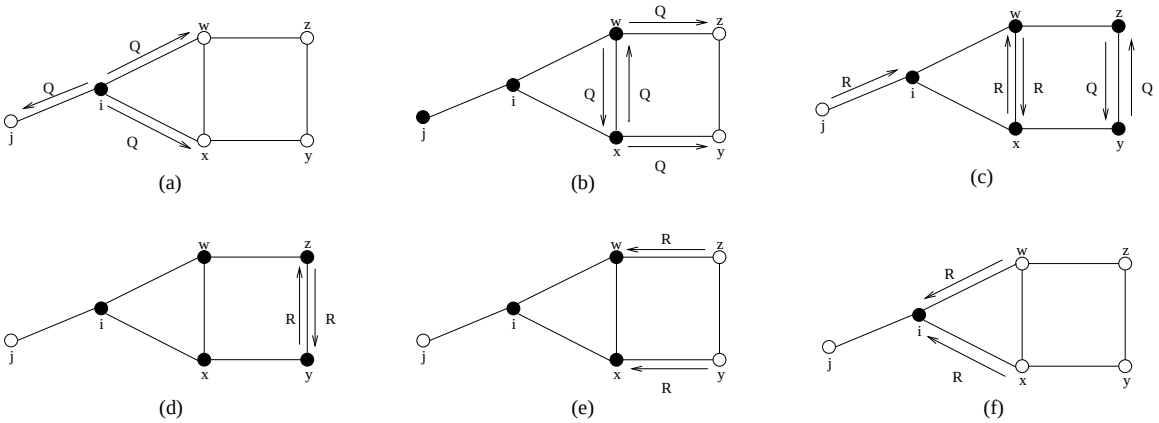

FIGURE 3.1: Normal action of reset algorithm. Filled circles denote active nodes

The core of the algorithm is the way in which queries are handled. When the node that receives a query is in passive state, it generally accepts the query, goes into active state, and sends queries to its neighbors. However, there are two exceptions to this rule. First, if the source of the diffusing computation is not the head of the link and the receiving node has a path to the link reported in the query, the node simply sends a reply. This prevents a diffusing computation originated by a node other than the head of the link from propagating to parts of the network where a physical path to the head of the link is known. Second, if the head of the link reported in the query receives it, the node sends a reply. If the content of the query it receives is different from the current link information, the head of the link also sends an LSU with a higher sequence number; this ensures that the correct information about the link will be known throughout the network.

Figure 3.1 illustrates the normal action for a diffusing computation concerning link (i, j). First, i sends queries to all its neighbors that are received at x, w, and j (Fig. 3.1(a)). These nodes go into active state and forward the query to its neighbors (Fig. 3.1(b)). Since neighbor w also received a query from i, it is active and immediately sends a reply to x after receiving the query from this node (and vice versa). y and z also forward the query to their neighbors (Fig. 3.1(c)). After nodes y and z also receive replies from each other (Fig. 3.1(d)), they return to passive state and send replies to their predecessors in the diffusing computation, x and w, respectively. After x receives the reply from y (Fig. 3.1(e)),

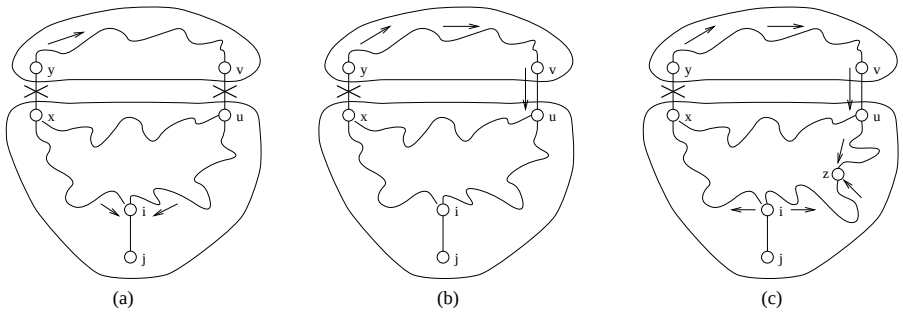

FIGURE 3.2: Detection of erroneous condition at node z

it returns into passive state and sends a reply to its predecessor in the diffusing computation, node i, as does w after receiving the reply from z (Fig. 3.1(f)). When node i receives the last reply, it also returns into passive state and the diffusing computation terminates.

In active state, the normal action taken by a node after receiving a query is to send a reply. However, if the node is active in a diffusing computation that was started at a node other than the head of the link, but the origin of the new query received is the head of the link reported in the query, then the new computation takes over. That is, the node becomes active in the computation started by the head of the link, and sends out new queries to all its neighbors. These queries ensure that the new diffusing computation also takes over at all other nodes that were active in the old diffusing computation.

The other exception to the normal processing of queries occurs when a node detects an erroneous situation when it is active in a diffusing computation originated by the head of a link and receives a second query originated by the same node for the same link, but such that the query contains different link-state information. This situation can only occur after a component of the network, in which an old diffusing computation has not terminated is reconnected to another component. This situation is illustrated in Figure 3.2, where node z is shown to receive two different queries. Because node z cannot decide which of the two diffusing computations is more recent, the situation must be corrected by the head of the link. Therefore, the node sends a reply that has an error tag, and tags its active state, meaning that all subsequent replies sent for the computation will be tagged as well. The propagation of the error tags ensures that the head of the link will be notified of the

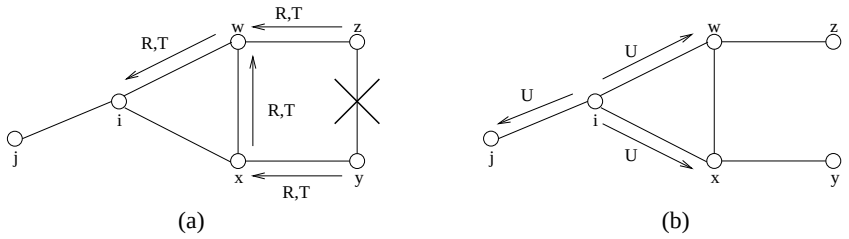

FIGURE 3.3: Propagation of tagged replies for diffusing computation to reset sequence number of (i, j) after link failure

erroneous situation, unless there is no physical path to the head of the link; in that case the diffusing computations both terminate and the information about the link will be erased in this part of the network.

When a link fails, the head of that link updates its topology table and sends a query for this link to its neighbors. If the node is active in a diffusing computation concerning another link and is still waiting for a reply to come over the failed link, then the node assumes that the reply has been received, and that this reply was tagged; this helps prevent deadlocks.

When the cost of a link changes or a new link is established, the head of that link initiates the flooding of an LSU for that link if the node is passive. However, if the head of the link is already active for the link when a change of cost or reestablishment of the link is detected, then the link must wait to distribute the LSU upon termination of the diffusing computation.

As described above, tags in replies are needed to signal an erroneous situation to the head of a link, who then sends LSUs with higher sequence number to its neighbors. Figure 3.3 (a) illustrates the propagation of tagged replies back to node i, the source of the diffusing computation. This example assumes that x received the query from w earlier than the query from i and that y and z are waiting for replies from each other when link (y, z) fails. After receiving the tagged reply from w and the reply from j, the head of the link (node i) sends LSUs with a new sequence number to its neighbors (Figure 3.3 (b)).

Unfortunately, the tagging mechanism may require extra communication in cases where it is not needed. In particular, whenever a link with outstanding replies fails, the reply is

assumed to be tagged, causing an unnecessary new LSU to be generated by the source (as in Figure 3.3). It is, however, possible to use some other means to signal the erroneous situation to the head of the link. For example, a different diffusing computation could be used to make sure that the information gets to the head of the link. This would increase the worst case complexity, but reduce the complexity in the more likely case of a link failure. Moreover, the correctness of the basic algorithm would not be affected if the new algorithm assures delivery of the needed information to the head of the link.

3.3.2 Reset for Selective Dissemination

To use the reset algorithm described in the previous section with LVA, some minor modifications need to be made. Since in LVA not all nodes need to store information about a given link, nodes that do not have information about a link (i.e. the link is neither in the topology table nor in the list of deleted links) need not participate in a diffusing computation concerning that link. Therefore, a node without information about the link (obviously, such a node is passive) simply sends a reply to the sender of the query, if the link-state information in the query does not cause the node to store the link. In addition, LSUs that arrive at a passive node are processed according to the LVA rules and not IFP's.

An active node that receives a query needs to process the link-state information in the query in addition to sending a reply. This is necessary to update the list of reporting nodes kept in LVA. A node that receives the last reply for a query must check for changes in the state of the link since it became active. With LVA, such changes can be caused by the buffered LSUs as well as by other information acquired during the active period; a change of the state of the link here includes more recent information as well as a switch from used to not used or vice versa. If any such change occurred, the appropriate *add* or *delete* update must be sent.

3.3.3 Fast Deletion of Old Information

An important feature of our reset algorithm is the fast deletion of old information. This is important, because reconnecting previously disconnected parts of the network can lead to significant overhead. For example, assume that some part of a network is disconnected. With aging, it takes a long time for the information about links in the other network component to be flushed. Therefore, if the sequence number of a link has been reset using premature aging and the network is reconstituted, it is possible for older information with a higher sequence number to pollute the network.

Figure 3.4 illustrates the above problem. In Figure 3.4 (a), an example topology is shown where a sequence number of 20 is known for link (i, j) throughout the network when link (x, y) fails. Figure 3.4 (b) shows the situation after node i initiated a diffusing computation to reset the sequence number of (i, j) to 0. In Figure 3.4 (c), the link between nodes x and y is reestablished before the old information with sequence number 20 expired in the disconnected component. As seen in Figure 3.4 (d), the obsolete information can now be propagated in the other part of the network as well. Although this situation will eventually be noted and corrected by the head of the link (node i), it may result in temporary routing loops.

On the other hand, if our reset mechanism is used rather than aging, the obsolete information in the component that was disconnected from the head of the link will very likely be erased by the time network reconststution occurs, because such erasure will occur in a matter of a few minutes. This also reduces the probability of temporary loops.

3.4 Correctness of Reset Algorithm

This section proves that the new reset algorithm is correct under selective dissemination of link-state information, which is a generalization of flooding. A subset of the same proof applies to topology broadcast.

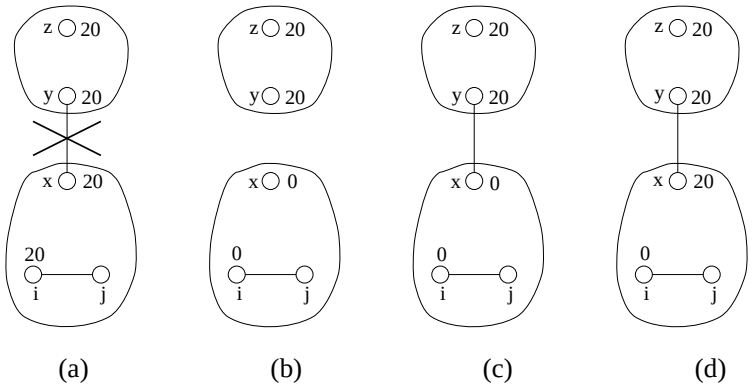

FIGURE 3.4: Example of polluting a network with old information.

A node is said to have a *notion of a link*, i.e., to have any information about a link, if it either has state information concerning that link in its topology table, or if it stores the sequence number of that link in its list of deleted links, but does not keep state information in the topology table. This definition would not be necessary if IFP were used, because the replication of complete topology information at every node implies that each node needs to know about every link. Message transmissions over an operational link are made reliable (i.e., messages are received without error and in the order in which they are sent) by means of a correct retransmission strategy between any two nodes across a link. With this assumption, the proof of correctness can simply assume, without loss of generality, that LSUs, queries, and replies are always sent reliably over an operational link. We also assume that the routers perform LVA error free. The reset algorithm is correct if, after a finite sequence of topology changes, any diffusing computation started at some node for a given link, terminates within a finite amount of time, and upon termination, the network has *consistent information* about the link.

Consistent information here has the following meaning:

- If the diffusing computation was initiated by the head of the link, then all nodes that have any notion of the link have the same information about it, and only nodes that use the link or whose neighbor uses the link have a notion of the link.

- If the diffusing computation was initiated by a node other than the head of the link, then the information about this link has been erased from those nodes that cannot reach the link.

The first step in proving correctness is to show that the reset algorithm terminates. Under the conditions stated above, we have the following:

Theorem 3.1 *Any diffusing computation for a given link* (i, j) *terminates within a finite amount of time.*

Proof: There are two possible scenarios for a diffusing computation not to terminate:

1. Deadlock could occur.

2. An infinite amount of queries could be generated.

To show termination of the algorithm, we need to show that neither of these scenarios can occur.

The proof that there can be no deadlock is by contradiction. Consider first a network with a static topology. Assume that there is some node x at which deadlock occurs. This implies that x does not receive a reply from at least one of its neighbors, say y. Node y must have some notion of (i, j) and y cannot be in active state when it receives the query from x, otherwise y would have sent a reply immediately after it received and processed the query from x. Hence, y becomes active with x's query and it must wait for a reply from a neighbor other than x, because x must send a reply to y. Following this line of argument, there must be an infinite number of nodes waiting for replies from nodes other than the node from which the query was received. This is not possible because the network is finite. Therefore, x cannot be in a deadlock situation.

Note that deadlock cannot occur even in a dynamic topology. This is the case because, when a link adjacent to x fails (or is established), then x simply assumes that a reply has been received over that link.

The proof that only a finite number of queries can be generated is also by contradiction. Assume that the diffusing computation does not terminate. Since there are only finitely many nodes, there must be a node x that produces an infinite number of queries. However, x cannot be the head of the link, which produces exactly one query. Therefore, node x must receive a query, send its own queries to all its neighbors, and send a reply infinitely often. Hence, at least one of its neighbors must do the same. Furthermore, w.l.o.g., this node must receive its first query earlier than x. In other words, there must be either a cycle of nodes which alternately go into active and passive state, or an inductive argument shows that there must be an infinite number of nodes in the stated situation. The first case can only occur if two parts of the network become disconnected and reconnect after the diffusing computation has terminated in one component but not in the other. Since there is only a finite sequence of changes in the network, this cannot go on indefinitely. Obviously, the second possibility contradicts the assumption of a finite network.

This concludes the proof of Theorem 3.1. **q.e.d.**

The algorithm works correctly if, after termination in a connected component, all the nodes in that component that have a notion of a given link have the same sequence-number (and the same link information) for that link. In addition, the information must be consistent as required by the underlying routing protocol, i.e., it must be up to date and, for LVA, correctly reflect whether it is used by the neighbor nodes.

This means that, in any part of the network that has no connection to a given link, the information about that link must be completely removed. In the part of the network that uses the link, the nodes that do have information about the link (this set is determined by the basic algorithm) have the latest sequence number for the link reported by the head of the link and the other nodes have erased any information about the link from their topology table.

Theorem 3.2 *Upon termination of a diffusing computation for a given link (i, j), the information about the link is consistent throughout the network, i.e., any node in the network has the correct information.*

Proof: The proof for this theorem is based on Lemmas 3.3 through 3.7. First, consider a connected stable network where the head of link (i, j) initiates a diffusing computation concerning that link. Then the proof for this theorem consists of two steps. First, Lemma 3.3 shows that, if a node needs information about a link, then it will receive a query with the correct information. Second, Lemma 3.4 shows that all nodes have consistent information when the diffusing computation finishes at the source of the diffusing computation.

Note that, after termination, the set of nodes that have any information about link (i, j) is a connected subset of the complete graph, because if a node has any information about (i, j), then there must be a path to i through nodes that also have information about (i, j). Also note that, as long as the diffusing computation is not terminated, i cannot produce any more recent information about (i, j).

Lemma 3.3 *A node x that has any notion of a link will receive a query with up to date information about (i, j).*

Proof: The proof is by induction on h, the minimum hop distance of x to i, the head of the link.

The base case is $h = 1$. In this case, x is a neighbor of i. When i initiates a diffusing computation, it sends queries with up to date information to all its neighbors. Since x is such a neighbor, it will receive a query.

For $h > 1$, assume that the lemma is true for all nodes with distance less than h.

Consider a node x with a distance of h hops to i. This node must have at least one neighbor y, that has a distance of $(h - 1) < h$ to i and that has a notion of (i, j). By the inductive hypothesis, y must have received a query with up to date information about (i, j). Since y has information about (i, j), it will send a query to all its neighbors, including x. At the same time, the diffusing computation cannot have terminated, since y did not receive a reply from x and therefore must be in active state. Hence, the query that x received from y contains up to date information. **q.e.d.**

Lemma 3.4 *When the diffusing computation terminates, all nodes have consistent information about link* (i, j).

Proof: The proof is by contradiction.

Note that a node that is in active state cannot send information that is not up to date to a neighbor.

Assume that node x does not have up to date information about (i, j) when the diffusing computation terminates. There are only two ways in which this could happen:

1. x received outdated information while it was in active state for the last time and buffered it, or

2. x received the outdated information after it sent a reply to its parent in the diffusing computation at the last time it was in active state.

For the first case to happen, there must be a neighbor y that first sends a reply to x's query and then sends the outdated information. Hence, y must be in the same situation as x. Following this line of argument, if the first case also applies for y, then y must have another neighbor in the same situation, and so on. Since the network is finite, we must find some node for which the second case applies; w.l.o.g., assume that y is such a node. For this case to happen, there are again two possibilities:

1. y must have a neighbor in the same situation, or

2. y had no information about (i, j) when it received the query that it replied to, and later got an outdated LSU from another neighbor, z.

Again, by an inductive argument and the fact that the network is finite, we can show that we must find some node for which the second case is true. So, we can again w.l.o.g. assume that the second case applies to y. For z, we again have two possibilities:

1. z is in the same situation as x, but obviously, the termination of its last active state happened earlier that for x, or

2. z did not receive any query in this diffusing computation.

The first case can be ruled out by an inductive argument and the fact that the network is finite; following the same line of argument, we must find an infinite sequence of nodes in the same situation described above for x, y, z. There cannot be cycles in these sequences, because a node in the same situation as z must finish its last active state earlier than the corresponding node in x's situation. The second case cannot occur, because z obviously must have information about (i, j) and, by Lemma 3.3 must receive a query before the diffusing computation terminates. Then, z would send a query to y, which in turn would send one to x. Hence, this was not the last time that x was in active state for this diffusing computation, which contradicts the assumption; therefore, the lemma must be true. **q.e.d.**

Now consider a network that is still connected, but where links can fail or come up while the diffusing computation is going on.

As long as no node thinks that it is not connected to (i, j), the diffusing computation still terminates, because a node assumes that it received a reply over a link that failed, and establishment of a new link does not cause additional queries to be sent. The assumptions for the proof of Lemmas 3.3 and 3.4 are not affected; therefore, the lemmas still hold in this case.

It remains to be shown that Theorem 3.2 holds if either the network becomes disconnected, a node (wrongly) perceives that it is disconnected, or the connectivity is reestablished after temporary disconnection. Lemmas 3.5 and 3.6 show that, all information about a link will be removed from the part of the network that does not have a path to the link, and that a diffusing computation that is started by a node that is seemingly disconnected does not produce incorrect link information.

Lemma 3.5 *If the network becomes disconnected, then all information about (i, j) will be removed from the part of the network that has no path to the link.*

Proof: Since a diffusing computation is started for any failing link, some node in the disconnected part will notice that it does not have a path to (i, j) and start a diffusing

computation. If no other node notices the disconnection early enough to start its own diffusing computation, then this computation will reach all nodes in the component in the network to which this node is connected. For this diffusing computation, the previous two lemmas hold, as well as Theorem 3.1 (termination). If multiple nodes in the disconnected component start separate diffusing computations, then:

1. Each of the diffusing computations terminates. A node that is already in active state for a diffusing computation originated at n, $n \neq i$, $n \neq m$ and receives a query for a diffusing computation originated at $m \neq i$ will simply send a reply, which is the same action a node that does not have information about (i, j) would take.

2. After termination, the nodes that were involved in at least one of the diffusing computations have consistent information.

3. All nodes that are disconnected from (i, j) will be involved in at least one diffusing computation. (If a node is disconnected from (i, j), and it does not receive any query, then it will start its own computation, since it will notice that it is disconnected.)

This shows the validity of the claim. **q.e.d.**

Lemma 3.6 *A diffusing computation started in a seemingly disconnected part of the network does not produce inconsistent link information.*

Proof: The rules of the algorithm prohibit a diffusing computation that is not originated by the head of the link to proceed through parts of the network that have a path to the link. Hence, such a diffusing computation will erase information only at those nodes that have no knowledge about a usable path to the link.

Furthermore, a diffusing computation for (i, j), originated at m, $m \neq i$, does not interfere with a diffusing computation initiated by i, since the latter simply overwrites the former.

Therefore, no permanent inconsistent information can be caused by a diffusing computation. **q.e.d.**

The last case to consider is that a new path to (i, j) is established to a portion of the network that was temporarily disconnected.

By the previous lemma, a diffusing computation started within the temporarily disconnected part will not cause inconsistencies in the network. The only possible problem could arise from a diffusing computation that was started at i, finished in the component connected to (i, j), but did not terminate yet in the part that was disconnected. (Lemma 3.1 still holds, so the diffusing computation will terminate eventually.) Lemma 3.7 shows that no permanent inconsistencies are produced in this case.

Lemma 3.7 *No permanent inconsistency is caused by the temporary disconnection and reconnection of a part of the network while a diffusing computation for any link is going on.*

Proof: In the case that the computation happens to travel over the new link that reestablished the connectivity, it will obviously reach the head of the link (i, j), unless another diffusing computation concerning the same link that started at i is going on.

If the diffusing computation reaches the head of the link, then node i will notice any possible inconsistency with the information stored in its topology table. It will send a reply (to avoid deadlock), and an LSU with the correct information if necessary. By the correctness of LVA [GB95], the content of this LSU will get to all nodes that need the information. Note that, due to the buffering of information received by an active node after the reply, the ongoing diffusing computation will not prevent the propagation of this information.

If another diffusing computation for the same link is going on, there will be some node x that is in active state from i when it receives a query for a diffusing computation started by i with different link-state information (if the two diffusing computations contain the same information, no problem arises). This node can not determine which of the two computations is the more recent. The problem of inconsistency must be solved by the head of the link by sending an LSU. To notify the head, a tagged reply is sent. Obviously, the head of the link will then receive a tagged reply, since nodes between the head of the link and x will send a tagged reply to their predecessor in the diffusing computation, if they

receive a tagged reply from one of their successors. Note that further topology changes do not change this property, since the assumed reply over a failing link is treated as tagged. **q.e.d.**

This concludes the proof of Theorem 3.2. **q.e.d.**

While the above theorems hold when LSUs and queries are sent one at a time and processed in order, to reduce communication overhead, it is desirable to combine multiple LSUs and queries together in packets. The following argument shows that this is possible, the algorithm still behaves correctly when such LSUs and queries are sent together and are processed out of order. For example, all the LSUs are processed first (as in the basic algorithm) and then the queries (one at a time, but in no particular order).

If a reply is sent in the same packet as an LSU or query concerning the same link, then it is crucial for the algorithm that the order of processing is correct, i.e., first the reply is processed and then the LSU or query.

There are two possible scenarios of failures due to out of order processing at the receiving node x:

1. x believes some query when it should not, or

2. x does not believe a query when it should.

Note that either case can only happen for a diffusing computation that was not started at the head of the link (which also always contains a delete LSU).

The first case implies that x has no path to the head of the link in its topology, while there is information in the packet that adds such a path. In this case, x does not have any notion about the link or is already involved in a diffusing computation. Either way, a reply is sent. Before x can use the link, it must receive an other LSU adding the link to its topology table.

In the second case, x does not believe a query because it still assumes a path to the head of the link, that is removed by other information in the packet. This does not pose a problem either, because after it realizes that there is no longer a path, x will start a diffusing computation concerning that link (and erase the link from its topology table).

3.5 Performance

3.5.1 Communication Complexity

For a single diffusing computation, the number of messages generated is $O(|E|)$. In the worst case, two queries are sent over each link, one in each direction. Note that there is exactly one reply sent for each query, which does not change the order of magnitude for the communication complexity.

The source of the diffusing computation obviously sends exactly one query over each outgoing link, because it must receive replies from all its neighbors, before it can send a second query, which is the condition for the computation to terminate. Now consider an arbitrary node other than the source of the diffusing computation. When this node receives a query, it either sends a reply to the sender, or it sends exactly one query over each outgoing link. Before it can send more queries, it must first receive replies from all its neighbors and then receive a new query. Hence, it must become active more than once for the same diffusing computation. This is not possible in the connected part of a network.

If IFP is used as the underlying protocol, the number of queries can easily be restricted to one per link. By not sending a query to the predecessor in the diffusing computation. With LVA, this extra query is used to update the set of reporting nodes at the predecessor node. On the other hand, with IFP, the worst case always occurs because the whole network is flooded with the information, while LVA produces fewer messages in the average case [GB95].

3.5.2 Time and Storage Complexity

In the connected component of the network, the worst-case time complexity is $O(x)$, where x is the number of affected routers. With IFP, x obviously is the number of nodes in the network, while with LVA it can be significantly less. The queries will travel to all nodes that do require the information, before the replies are sent back on the same paths and with

the same time complexity. In the worst case, all affected routers lie along a single path, causing $O(x)$ communication steps.

Computational complexity at routers is determined by complexity of underlying protocol, for each query, only constant work is added to that already being done.

The extra storage required while a node is passive is constant, only an extra tag indicating the state is needed. While a node is active, in addition to some extra state information, $O(|N_x|)$ storage (where $|N_x|$ is the set of neighbors of node x) is required to keep track of the received replies at node x, and to buffer LSUs.

3.6 Summary

We presented a new algorithm to reset sequence numbers in routing protocols. This reset algorithm, which is based on a recursive query-response process, makes it possible to use a bounded sequence-number space without a need for periodic retransmissions or aging. Thus, its time complexity is determined entirely by the time it takes to traverse the network, and it does not rely on any global timers.

The reset algorithm can be used with routing protocols based on flooding as well as selective dissemination of link-state information to speed up their convergence. For instance, using our reset algorithm in OSPF, even when resources fail, all link-state information will be distributed in time proportional to the time needed to traverse the network, which should take in the order of minutes at the most. A version of intelligent flooding based on this reset mechanism was introduced.

We have shown that the reset algorithm leads to a correct routing protocol when applied to the selective dissemination of link-state information, which is a generalization of flooding.

3.7 Appendix: Pseudo Code for Reset Algorithm for Flooding

Notation

TT_x Topology table at node x, entries (i, j, l^i_j, sn, r, d), where
- (i, j) Link from node i to node j
- l^i_j Length of link (i, j)
- sn Sequence number of link
- d Status for diffusing computations: active/passive, set of replies received, source, predecessor

ST_x Shortest path tree at node x.
N_x Set of neighbors at node x.
t Sequence number at node.
t_j Last sequence number of neighbor j.

Messages are (ordered) sets of updates of the form
$(i, j, l^i_j, sn, type, ds)$ for link (i, j) with cost l^i_j, where
- sn Sequence number
- $type$ Type of update: update, query, reply
- ds Source of diffusing computation (if applicable)

Process Update Packet

Process update packet receive from neighbor. First process the replies contained in the packet, then the updates, and at last the queries. When done with this, assemble the packets to be sent to the neighbors.

```
procedure process_packet (x, n, packet)
-- Parameters:
-- x: name of node that executes procedure
-- n: name of node that sent packet
-- packet: packet to be processed
  begin
     process_replies (x, n, packet)
     process_updates (x, n, packet)
     for all q ∈ packet -- query q
        process_query (x, q)
     end for
     assemble_and_send_new_packets (x)
  end process_packet
```

Process Updates

Process all updates in packet. If the node is active for the link described in an LSU, the update is buffered; if the node is passive, the update is processed according to the rules for the flooding algorithm.

```
procedure process_updates (x, message)
- - Parameters:
- - x: name of node that executes procedure
- - message: packet to be processed
  begin
    for all m = (i, j, l^i_j, sn, update) do
      if TT_x(i, j).d = active then
        if reply from message.source received then
          buffer m
        else
          discard m
        end if
      else - - passive
        if (i, j) ∈ TT_x then
          if TT_x(i, j).sn < m.sn then
            if m.l^i_j < ∞ then
              TT_x(i, j) = m
            else
              TT_x(i, j) = ∅
            end if
          else if TT_x(i, j).sn > m.sn then
            send (message.source, (TT_x(i, j), update) )
          end if
        end if
      end if
    end for
  end process_updates
```

Process Replies

Process all replies in update packet. If the reply is the last one expected, go into passive state, process buffered updates, and send a reply to the predecessor in the diffusing computation.

```
procedure process_replies (x, n, packet)
- - Parameters:
- - x: name of node that executes procedure
- - n: name of node that sent packet
- - packet: packet to be processed
  begin
    for all rep ∈ packet do - - rep = (i, j, t)
      if TT_x(i, j).d = active then
        TT_x(i, j).d.received = TT_x(i, j).d.received ∪ n
        if t = true then
          TT_x(i, j).d.tag = true
        end if
        if TT_x(i, j).d.received = N_x then
          - - all replies received
          if x = i then
            if TT_x(i, j).d.tag = true then
              for all k ∈ N_x do
                send (k, (TT_x(i, j), update)
              end for
            end if
          else
            new_reply = (i, j,TT_x(i, j).d.tag)
            send (TT_x(i, j).d.predecessor, new_reply)
          end if
          TT_x(i, j).d = passive
          if there are buffered updates for (i, j) then
            for all m = (i, j, l^i_j, sn, update) in buffer do
              if m.sn > TT_x(i, j).sn then
                TT_x(i, j) = m
                for all k ∈ N_x do
                  send (k, m)
                end for
              end if
            end for
          end if
        end if
      else
        discard rep
      end if
    end for
  end process_replies
```

Process Query

procedure process_query (x, q)
– – Parameters:
– – x: name of node that executes procedure
– – query $q = (i, j, l^i_j, sn, type, ds)$
  begin
    if $TT_x(i,j).d$ = passive **then**
      if $x = i$ **then**
        send $(n, (i, j, reply))$
        send $(n, (TT_x(i,j), update)$
      else if $q.ds = i$ **or** $i \in ST_x$ **then**
        – – source of query is head of link or there is a path to the link
        set_entry $(TT_x, i, j,$ active, $i, q.l^i_j, 0)$
        $TT_x(i,j).d$.predecessor $= n$
        for all $k \in N_x$ **do**
          send $(k,$ New_query$)$
        end for
      else – – source of query $\neq$ head, path to link
        send $(n, (i, j, reply))$
      end if
    else – – (i, j) active
      if $q.ds = i$ **then** – – source of query is head of link
        if $TT_x(i,j).d$.source $= i$
            and $TT_x(i,j).l^i_j = q.l^i_j$ **then** – – same diffusing computation
          send $(n, (i, j, reply))$
        else if $TT_x(i,j).d$.source $\neq i$ **then** – – diffusing computation from other source
          set_entry $(TT_x, i, j,$ active, $i, q.l^i_j, q.sn)$
          $TT_x(i,j).d$.predecessor $= n$
          for all $k \in N_x$ **do**
            send $(k,$ New_query$)$
          end for
          updated = true
        else – – special case: different diffusing computation from head of link
          if $x = i$ **then** – – received at head of link
            send $(n, (i, j, reply))$
            send $(n, (i, j, l^i_j, sn, update))$
          else
            $TT_x(i,j).d$ = active, tagged
            send $(n, (i, j, reply))$
          end if
        end if
      else – – source of query different from head of link
        if $TT_x(i,j).d$.source $= i$ **then** – – active from head
          discard query
          send $(n, (i, j, reply))$
        else – – active from other node
          process update part
          send $(n, (i, j, reply))$
        end if
      end if
    end if
  end process_query

Assemble New Packets

Assemble new packets to be sent to the neighbors. This procedure also detects unreachable links. The send-procedure used in other procedures should be implemented as buffering, then the buffered information is added to the packages here.

```
procedure assemble_and_send_new_packets (x)
– – Parameter:
– – x: name of node that executes procedure
  begin
    for all (i, j) ∈ TT_x do
      if TT_x(i, j).d = passive and i unreachable then
      – – i unreachable is the same as i ∉ ST_x
        TT_x(i, j).d = active
        TT_x(i, j).d.source = x
        u_message = u_message ∪ (TT_x(i, j), query, x)
      end if
    end for
    for all k ∈ N_x do
      message = u_message ∪ buffered information for k
      send (k, message)
    end for
  end assemble_and_send_new_packets
```

Set Entry of Topology Table

```
procedure set_entry (TT_x, i, j, status, source, l, sn)
– – Parameters:
– – TT_x: topology table of node that executes procedure
– – i, j: head and destination of link
– – status: status of link (active/passive)
– – source: source of diffusing computation
– – l: cost of link
– – sn: sequence number of link state
  begin
    TT_x(i, j).d = status
    TT_x(i, j).d.source = source
    TT_x(i, j).l^i_j = l
    TT_x(i, j).sn = sn
  end set_entry
```

Link Cost Change

This procedure is called by the underlying protocol when the cost of a link changed. It buffers the information if the link is active; otherwise the information is processed and updates are sent to the neighbors.

```
procedure link_change (x, y)
– – Parameters:
– – x: name of node that executes procedure
– – y: name of destination of link
  begin
     if TT_x(x, y).d = active then
        buffer update {(x, y, l_y^x, sn, update)}
     else
        process_updates (x, {(x, y, l_y^x, sn, update)})
        assemble_and_send_new_packets (x)
     end if
     sn = sn + 1
  end link_up
```

Failure of Link

This procedure is called by the underlying protocol when the link to a neighbor failed. It buffers the information if the link is active; otherwise the information is processed and updates are sent to the neighbors.

```
procedure link_failure (x, y)
– – Parameters:
– – x: name of node that executes procedure
– – y: name of destination of link
  begin
     N_x = N_x − {y}
     if TT_x(x, y).d = active then
        buffer update {(x, y, ∞, sn, update)}
     else
        process_updates (x, {(x, y, ∞, sn, update)})
        assemble_and_send_new_packets (x)
     end if
     sn = sn + 1
  end link_failure
```

Link Comes Up

This procedure is called by the underlying protocol when a new link to a neighbor is detected. It buffers the information if the link is active; otherwise the information is processed and updates are sent to the neighbors.

```
procedure link_up (x, y)
-- Parameters:
-- x: name of node that executes procedure
-- y: name of destination of link
  begin
      N_x = N_x ∪ {y}
      process_updates (x, {(x, y, l_y^x, sn, update)})
      assemble_and_send_new_packets (x)
      sn = sn + 1
  end link_up
```

Chapter 4

Hierarchical LVA

4.1 Introduction

Although we have shown that LVAs are more scalable than LSAs and DVAs, using LVAs with a flat addressing structure is not sufficient for a net to scale to very large numbers of nodes and destinations. Any routing algorithm that requires routers to know about every single destination in an internet, becomes infeasible as the internet grows. The storage requirements as well as computational and communication overhead become too costly. To address this problem, the amount of information stored and communicated must be reduced using address aggregation schemes.

The goal of any address aggregation scheme is to reduce the size of the topology databases or routing tables kept at routers, thereby reducing the amount of data that needs to be communicated, processed, and stored. The main idea in aggregation schemes is that a router keeps in its database one entry per node or link that is "close," and an entry for a set of nodes or links further away [Kam76]. To achieve this, hierarchies of addresses are formed by grouping together ("clustering") nodes that are close together.

The OSPF [Moy94] and ISO IS-IS [ISO89] protocols define areas that correspond to well defined portions of an internet. Areas are defined statically, and to route traffic among such areas, a backbone is used to interconnect all areas. In OSPF, all inter-area traffic must be routed via the backbone.

There have been many hierarchical routing proposals described in the past based on the notion of areas, which are also called clusters [Ste95]. The first such proposal was McQuillian's [McQ74]; this proposal was analyzed in detail by Kamoun and Kleinrock [KK77]. Most prior proposals on hierarchical routing have routing algorithms based on topology broadcast or variations of the distributed Bellman-Ford algorithm. Ramamoorthy et al. [RT83, TRTN89] proposed an algorithm based on link-state information for hierarchical routing. According to this algorithm, a node maintains complete topology information of each area to which the node belongs, and the topology of an area at a given level is given by the interconnection of the lower-level areas within it. More recently, Murthy and Garcia-Luna-Aceves [MG97] proposed an area-based hierarchical routing algorithm called HIPR that is based on McQuillian's clustering scheme and the loop-free path finding algorithm [GM97] which is a loop-free algorithm based on distance vectors.

In this chapter, we introduce a new area-based hierarchical routing scheme that uses LVA as its basic routing algorithm. This new scheme, which we call area-based link-vector algorithm (ALVA) supports multiple levels of hierarchy and does not rely on a backbone for inter-area routing. ALVA allows more flexible topologies and shows improved performance by removing the bottleneck backbone. The main motivation for this new scheme is to provide an approach based on link-state information that does not require complete topology information for each hierarchical level. As we show subsequently, it constitutes the basis for developing internet routing protocols based on link-state information that are much more scalable than OSPF.

Section 4.2 describes the hierarchical routing algorithm. Section 4.3 proves its correctness. Section 4.4 discusses its complexity and presents simulation results addressing its average performance.

4.2 Area-Based LVA (ALVA)

To allow a hierarchical structure, the basic network model described in chapter 2 is extended. Nodes of the graph are clustered into subgraphs called areas. Although the new hierarchical

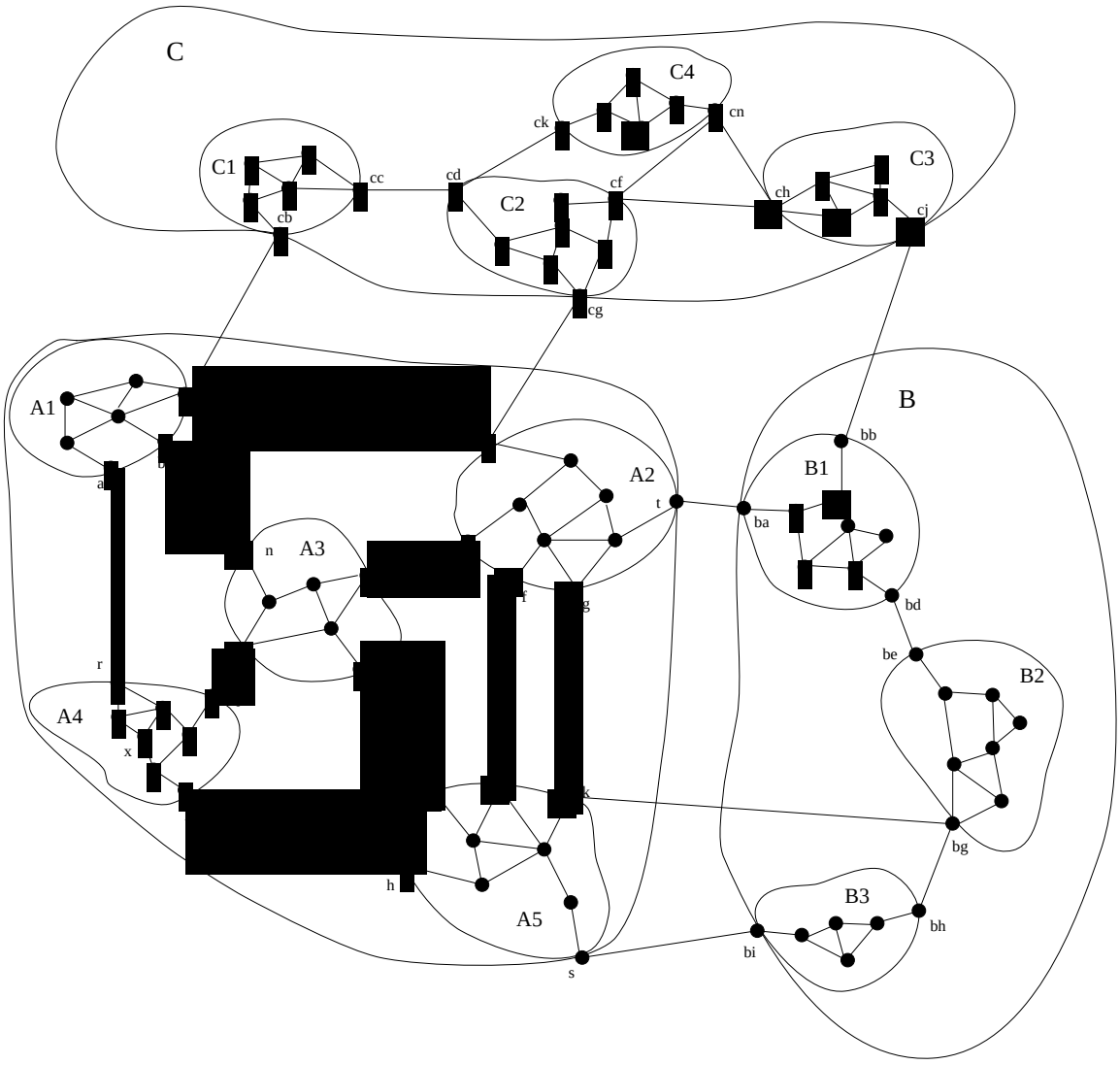

FIGURE 4.1: 2-level hierarchical network

routing algorithm can be used with overlapping clusters with only minor modifications, for simplicity, we assume that the areas are disjoint, i.e. every node belongs to exactly one area.

According to ALVA, nodes are clustered into areas organized into multiple hierarchical levels, so that areas can be grouped into higher-level areas as well. Figure 4.1 shows an example topology with three levels of hierarchy. Links in this topology are assumed to be bidirectional, with unit cost in both directions. The nodes (named in lower case) make up

level 0 in the hierarchy. Level 1 consists of the areas A1..A5, B1..B3, and C1..C4, while we have the areas A, B, and C at the top level, level 2. In this example topology, only border-nodes are named, with the exception of node x, which is an interior node of area $A4$.

A *border node* is a node that has a link to a node that belongs to a different area. A k-level border node is a node that connects k-level areas. Nodes can determine to which area a given address belongs, and at which level of the hierarchy two given addresses differ. With this, nodes can dynamically determine whether they are border nodes (this may change with link failures or establishments,) and at which level their border is. The basic operation of ALVA is as follows:

- For routing within an area, flat LVA is used.
- For inter-area routing, LVA is applied on the topology representing the connectivity among areas at any particular level.

At any given level, shortest-path routing is used among all areas that are contained in the same area one level up in the hierarchy. Because areas are seen as single entities by remote routers, the cost to traverse them cannot easily be determined; since the cost of the links between the areas is outweighed by the area traversal cost, using the actual for those links need not improve overall performance of the algorithm. Accordingly, for simplicity, we use minimum hop routing across areas in this chapter.

The pseudo code in the appendix (section 4.6 provide a formal specification of ALVA. The following sections are used to describe the information stored and communicated, as well as ALVA's operation, in more detail. For simplicity, we assume that the sequence numbers used to validate updates are based on unbounded counters. In practice, a mechanism using a finite sequence number space must be used.

4.2.1 Information Maintained at Nodes

With respect to the information exchanged, all routers act as peers in ALVA. This does not mean that the information stored is the same at all routers, but that the type of information

is the same. There are no special routers that need to store any additional information. Thus, we can ensure that any routers can, without delay, accept the additional functionality of a border router if a new link crossing a border is established.

Each router maintains a topology table and a source graph. The latter is used to derive the routing table. The topology table may be viewed as being split up into one table for each level in the hierarchy (as is assumed in the pseudo code in the appendix;) this is merely an implementation matter for the path-selection algorithm.

In principle, the topology table contains the following information about all links known to the router, and belonging to the router's own 1-level area: head and destination of link, cost of link, sequence number, and the list of its reporting nodes. Again, the reporting nodes of a known link are those neighbors of the router who have notified using that link. If more than one routing policy is used in the network, multiple costs can be reported for the same link.

For inter-area links, additional information must be stored. Because an inter-area link represents connectivity rather than a particular physical link, it may be that this link actually corresponds to multiple links. Thus, checking whether an update concerning such a link is recent becomes a problem, given that there can be no unique sequence number assigned to it. To solve this problem, the sequence number with the head of the actual link and store the ID of the head of the link together with it. Different neighbors may report different heads about the same inter area connections to a node. The node then stores all the different heads concerning the connection, but forwards only one of them. To reduce communication overhead, all nodes should use the same criterion as to which head to report in such a case, but this is not required for the protocol to work correctly.

Of course, the list of reporting nodes must be kept on a per head of link basis as well. The list of reporting nodes can easily be stored as a bit vector, since only neighbors of a node can be in that list. Thus, the storage overhead of that list is relatively minor.

```
level 0:                    level 1:                    level 2:
   all links for paths         A1 - A2, c                  (none)
     x -> r                    A3 - A2, m                may store
     x -> q                    A5 - A2, j (, k)            A - B, s (, t)
     x -> p                    A2 - B,  t                  A - C, c (, d)
     (and other paths          A5 - B,  s (, k)
      within A4)               A1 - C,  c
                               A2 - C,  d
     p - A5
     q - A3                  may store
     r - A1                    A4 - A1, r
                               A4 - A3, q
                               A4 - A5, p
```

FIGURE 4.2: Topology at Node x

The source graph contains all links that are used on a preferred path to any destination, as determined by the local path selection algorithm. In the case of shortest path routing, it is simply the shortest paths tree.

Figure 4.2 shows a textual representation of the links known at node x. Figure 4.3 shows a graphical representation of the topology databases at node x and the border node k.

As can be seen in Figures 4.2 and 4.3 (a), x knows all the links necessary for it (or one of its neighbors) to reach any destination within the level 1 area $A4$. In particular, it knows all the links necessary to reach the border-nodes. In addition, the local table contains links from these border-nodes to the neighboring level 1 areas. (The internal topology of $A4$ is too small to show any significant saving in space as compared to topology broadcast here. However, it should be noted that a few of the links are known only in one (the "useful") direction, exhibiting some of the savings due to LVA. At the next level of hierarchy, the figures show a partial view of the inter-area topology. Note that, while node x sees only one way to each of the level 2 areas B and C, border-node q in the same area actually knows about the alternatives through area $A2$, enabling it to react fast to changes in the topology and then propagate that information within its area.

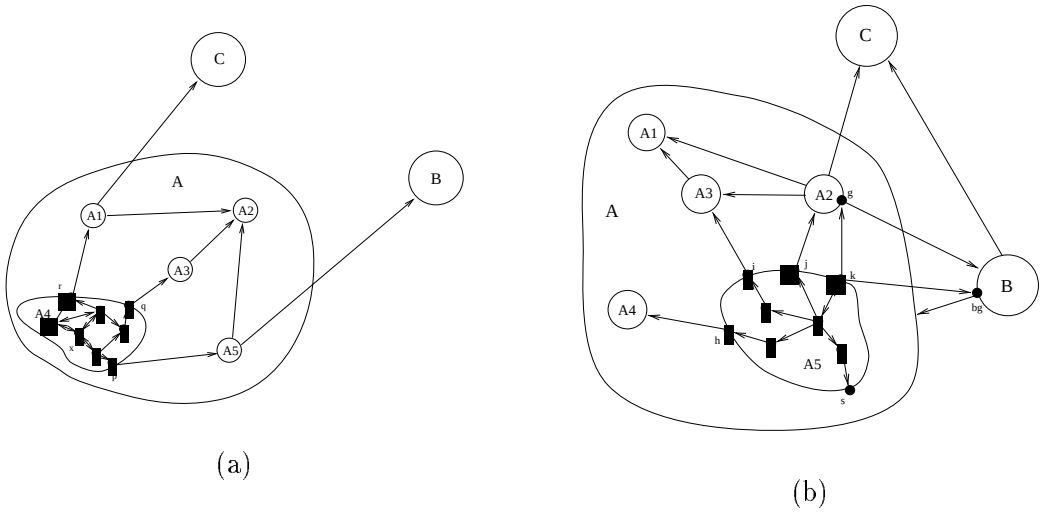

FIGURE 4.3: Topology at Nodes x (a) and k (b)

Although the information concerning links leaving the own area in levels 1 and 2 is redundant, it may be beneficial to store it in the tables to simplify the path selection algorithm.

Figure 4.3 (b) shows the topology as seen by border-node k of area $A5$. It can be seen that the two topologies are quite different. However, due to the way the tables are formed, these differences cannot create any routing loops. Note that, by virtue of being a border-node to area B, k actually knows about the connection between areas B and C. It does not propagate this information within its area though, because it prefers the path through $A2$ to reach C.

4.2.2 Information Exchanged between Nodes

While there is no difference in the data stored at nodes, the information exchange within an area is obviously different from the inter-area exchange. Border nodes filter the information that is forwarded across their borders. They do not forward internal information about their area across the area border, but add the appropriate head of link information to updates concerning links leaving their area.

```
forwarded to q:                    forwarded to neighbors in A3:
     A3 - A1, n                         o  - A4
     A3 - A2, m                         A1 - C,  c
     A3 - A5, l                         A5 - B,  s
     A1 - C,  c                         links used within A3
     A5 - B,  s
```

FIGURE 4.4: Link-states forwarded by Node o

Whenever there is a change in a node's source graph, it sends incremental updates about the change to its neighbors: it sends an *add* update for links that they are using to get to any destination; it sends a *delete* update for links that they used before but that are not used any more. For links within an area, each such update contains the cost, the sequence number, and the type. Updates concerning links crossing area borders contain the areas of origin and destination, a sequence number and the head of the link reporting that sequence number.

Border links originating in the own area are propagated with the actual head of the link, but with an area address as destination by the border node.

A border node makes sure that no link from within its area is reported to its peer in the other area. Links to other areas are converted to a hierarchical form and one of the actual heads of that connectivity is chosen to propagate the respective sequence number.

To illustrate the differences in how information is forwarded by border-nodes, Figure 4.4 shows which link states are forwarded by node o. Since o must distinguish between the recipients of its update packets and assemble different packets to neighbor q and its interior neighbors.

The differences are further illustrated in Figure 4.5, where the links that are forwarded by node h to its neighbor p, which is in a different area at level 1, and the links that s forwards to its neighbor bi over a level 2 boundary.

```
h forwards to p:
     A5 - A2, j
     A5 - A3, i
     A5 - A4, h
     A5 - B,  s
     A2 - C,  d
```

```
s forwards to bi:
     A - B, s
     A - C, d
```

FIGURE 4.5: Link-states forwarded by Node h and s over area boundaries

4.2.3 Operation of ALVA

The operation of ALVA is very similar to that of basic LVA. When an update message is received from a neighbor, every update in the message is examined and the topology table changed as necessary.

First, consider that the update is an *add*: if there is no information about the link in the topology table, then the link is added to it. If the link is already present in the table, then its value is changed if the sequence number indicates a more recent update. If the sequence number is the same as the one stored, then the neighbor that sent the update message is added to the list of reporting nodes.

In the case of a *delete* update, the sender of the message is removed from the set of reporting nodes and. if the set becomes empty, the link is removed from the topology table.

In either case, an update containing recent information is sent back to the neighbor who sent the message if an update is found to be out of date (i.e., its sequence number is smaller than the one stored.)

If there was any change in the topology table, then the updated topology table is used to obtain the new source graph, using the local path selection algorithm. From the source graph, the routing table is updated. At last, the new source graph is compared with the previous one to assemble the update packets that are sent to the neighbors. In principle, an *add* update is generated for any new link in the source graph, and for any link whose sequence number changed as a result of the update procedure. For any link that was previously part of the source graph but is no longer being used, a *delete* update is generated. Of course, a border node must filter the propagation of this information as described above.

The main difference between ALVA and the basic LVA lies in the fact that the sequence numbers and reporting nodes for inter-area links are updated on a per head-of-link basis in ALVA. This also means that, if the head of the link changed for some inter-area link, two updates must be generated, one to delete the old head and another to add the new one. Because all routers use the same criterium to choose which head to advertise, this is a rare occurrence.

4.3 Correctness of ALVA

The proof of correctness for ALVA assumes that update messages are transmitted reliably and received and processes in the order that they are sent. In addition, we assume that there is a finite number of changes in link state up to time t_0, after which time there are no more changes. With these assumptions, the following theorem shows that ALVA is correct.

Theorem 4.1 *After a finite time after t_0, no more updates are sent in the network, and all routers have up-to-date link-state information in their topology table and have computed correct hierarchical source graphs.*

The proof of correctness consists of two parts: first, it is shown that ALVA terminates. The second part shows that the information in the network is consistent upon termination and thus correct routes have been computed.

Both parts of the proof build on the properties proven for LVA [GB95] and extend the proofs for LVA to the hierarchical algorithm. The following lemmas constitute the proof.

Lemma 4.2 *Area-based LVA terminates within a finite amount of time after t_0.*

Proof: The proof that the hierarchical LVA terminates is by induction on the number of levels (k) in the hierarchy. In each inductive step, the proof is by contradiction.

The base case for the induction is a topology with a one-level hierarchy ($k = 1$).

The proof for LVA assumes that an infinite number of updates (*add* of *delete*) is generated for some link, and it is shown that this is impossible due to the finite number of nodes in

the network and the fact that the node detecting the link change sends exactly one update. Because flat LVA is used within the lowest-level areas and no information is propagated outside an area concerning topology changes within the area, it is clear that the algorithm terminates for such changes.

It remains to show that the algorithm terminates if there is a change in the connectivity between two areas. Note that all nodes use LVA on the graph comprising all inter-area links in the network. The exact same argument used for flat LVA can now be used: assume that an infinite number of updates is generated. This implies that there is at least one node that generates an infinite sequence of updates about some link l. In turn, this implies that a neighbor of this node also generates an infinite sequence of updates concerning the same link. The proof for LVA proceeds showing that there must be an infinite sequence of nodes who start sending infinitely many updates caused by that same change. However, this argument is only valid if nodes can validate updates, which can be accomplished by sequence numbers. Here lies the only difference in the proof. Because those links can be detected at multiple heads-of-link, it must be assured that this does not lead to a "flip-flop-effect," where nodes switch between the heads-of-link whose sequence number they use for reporting the link. This problem does not apply to the two areas that are actually connected, in these areas the links are reported with their physical heads-of-link, inclusion of these links does not alter the termination property of LVA within areas. In remote areas, if we require that all nodes who receive multiple heads choose the head that they propagate using the same criteria (for example, using the smallest address,) then no infinite sequence of adds and deletes will be created. Hence, ALVA terminates when there is one level of hierarchy.

Now consider a topology with $k > 1$ levels of hierarchy. Assume that ALVA terminates for $k - 1$ levels of hierarchy.

A k-level hierarchical topology is composed of $(k - 1)$-level areas and links connecting these areas. By the inductive hypothesis, we know that ALVA terminates for any changes within the $(k-1)$-level areas. It remains to show that ALVA terminates if there is a change

in the connectivity between two k-level areas. The argument here is very similar to the case of one level of hierarchy. Again, LVA is used at the k-th level of hierarchy, and the only difference to the flat case is that there can be multiple heads-of-link, that could cause the described "flip-flop-effect." Again, this is prevented by requiring the consistent choice of the head-of-link that is reported to a neighbor. Hence, ALVA terminates in a k-level hierarchy. **q.e.d.**

Lemma 4.3 *Within finite amount of time after t_0, all routers have the consistent information necessary to compute correct source graphs.*

Proof: The proof that all nodes have consistent information when ALVA terminates is by induction on the levels in the hierarchy.

Again, the base case for the induction is a one-level hierarchy.

The proof that information is consistent in a finite time after topology changes cease for LVA is by induction over the length of paths in hops. In a similar fashion, we can argue the same case in the higher level.

Within any area, LVA is used. Therefore, a any node – in particular any border node – in the area has correct information about the topology within the area that it is part of. In addition, it knows about all links that it needs to route to neighboring areas. (A neighboring area appears as a destination in the level 0 topology table, therefore, every node – including border-nodes – knows at least one link to that destination.) Since a border node forwards the latter information to its neighbors in the other nodes, the border-nodes of these areas know about the connectivity of their neighboring area at level 1. This is the base case, one hierarchical hop. The information is propagated within the neighbor-area, using LVA rules for a network at level 1, where minimum hop routing is used for the computation of the preferred paths. From the correctness of this (flat) LVA at the higher level follows the correctness of the hierarchical scheme.

The formal proof for this argument uses induction, as in the flat case. The base case – neighboring areas – is described above. Then, for $h > 1$ hops assume that the correct, needed information to reach a destination is known in areas that are less than h hierarchical

hops away from this destination. Consider a path that spans h hierarchical hops. Then, we know that there is a flat path to the first area on that path. This area is $h-1$ hierarchical hops away from the destination. The subpath from that area to the destination must be optimal, hence, by the inductive hypothesis, it is known in that area. But since it is used and known, it must be propagated to all the neighboring areas by its border-nodes, and therefore also be known in the area we first considered. This, together with the known path to that neighboring area, means that the complete path is known h hierarchical hops away.

Now consider the case that we have $k > 1$ levels of hierarchy. Assume that the algorithm yields consistent information for $k-1$ levels.

A k-level area is composed of $(k-1)$-level areas. By the inductive hypothesis, all nodes have consistent information about their $(k-1)$-level. In addition, the links between $(k-1)$-areas, as well as their connectivity to outside areas is known to all $(k-1)$-level border-nodes. Hence, for a given k-level area, this information is also known at all its border-nodes, since a k-level border-node is also a border-node at levels $1, \ldots k$. Then, we can use the same inductive argument as in the base case, using the links between k-level areas as hierarchical hops.

This proves Lemma 4.3. **q.e.d.**

4.4 Performance

Since LVA has been shown to outperform the ideal link-state algorithm in [GB95], it can be expected that ALVA performs better than area-based schemes based on flooding, such as OSPF, by reducing the control traffic both within areas as well as across the backbone. To verify this expectation, we compared ALVA with OSPF in several simulations. Simulation were performed using random graphs with 100 nodes. Nodes had an average degree of approximately 3. Recent work [GR97] shows that this is a realistic node degree for internetworks. The topologies were produced according to two general schemes. According to the first scheme, there is a backbone with 56 nodes, one area with 30 nodes, and 14 stub areas with one node each. We chose to use stub areas because we were particularly

interested in the effect of changes in the backbone. This topology type allows us to have many destination areas but to focus on the effects that a change in the backbone has within the backbone and in the complete area. In the second scheme, the backbone contains 40 nodes and there are four areas with 15 nodes each.

To obtain random topologies according to these schemes, for each area (including the backbone) nodes are places randomly in a plane. Any two nodes u, v within the area are then connected according to the exponential model as proposed by Zegura et al. [ZCB96], i.e., with probability

$$P(u, v) = \alpha e^{\frac{d}{L-d}},$$

where d is the Euclidean distance between u and v, L is the maximum distance between any two nodes, and $0 < \alpha \leq 1$ is a parameter of the model. In addition, we make sure that all areas are connected graphs. Then, each area is connected with the backbone at two randomly chosen nodes.

This method to obtain topologies allows us to study networks that exhibit the characteristic of the logical star configuration that OSPF requires for inter-area traffic [Moy94].

To simulate OSPF, we make the following assumptions:

- Areas contain exactly one mask, i.e., they are seen as a single entity from outside the area. In terms of storage and communication overhead needed, this actually presents the best case for OSPF.

- Border nodes belong to exactly one area and the backbone. A border node runs two copies of the flooding algorithm, one for the backbone and one for the area to which it belongs.

- A border node reports to the backbone that it has a link to the area of which it is part.

- A border node reports within its area links to all other areas with costs as determined by the shortest path algorithm in the backbone topology.

We evaluate the performance in terms of update messages sent and number of steps required for the algorithms to converge. When a node receives an update, it compares its local step counter with the sender's, takes the maximum, and increments the counter. This way we obtain the number of sequential update message exchanges between neighbors needed for convergence. In addition, we compare the size of the topology tables stored by the routers. In terms of these criteria, the assumptions stated above actually represent the best case for OSPF.

For our simulation, we assume that control packets are transmitted error free and are processed one at a time in the order received. OSPF and protocols based on ALVA provide their own retransmissions. Using equivalent mechanisms, ALVA requires less overhead than topology-broadcast to ensure reliable transmission of updates. Packets sent over failed links are dropped. To detect new connectivity or link failures, a simple hello protocol was used (much like in the OSPF specification.)

Figures 4.6 through 4.10 show the results of our simulations. Results are shown for changes in link cost, link failures, link establishments, node failures and node establishments. The bars represent the average (mean) number of messages and steps, respectively, while the markers show the standard deviation. To obtain these results, we performed the changes for every single link and node in the network. Thus, no sampling errors need to be presented.

Figures 4.6 and 4.7 show the overall results for one representative topology of each class. These results include changes at the borders as well as changes in the backbone and other areas.

In Figures 4.8 and 4.9, more detailed results are shown. The left graph in figure 4.8 shows the message sent for changes within the backbone, while the right graph in that figure represents changes within the other area of the topology of the first type. Similarly, Figure 4.9 shows the number of messages until convergence for changes within the backbone and in one of the areas for the topology of the second type.

It is clear that ALVA needs less time and fewer messages to converge for changes in single links in the backbone as well as the areas. OSPF behaves better only when nodes

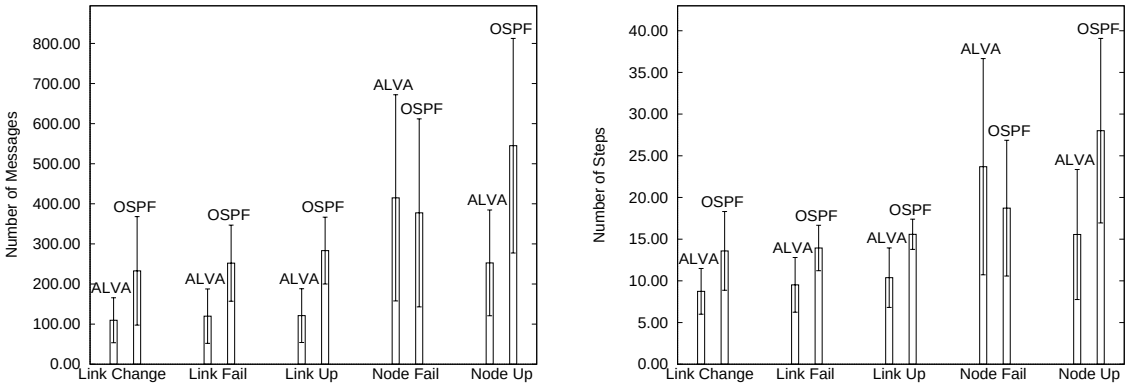

FIGURE 4.6: Topology type 1

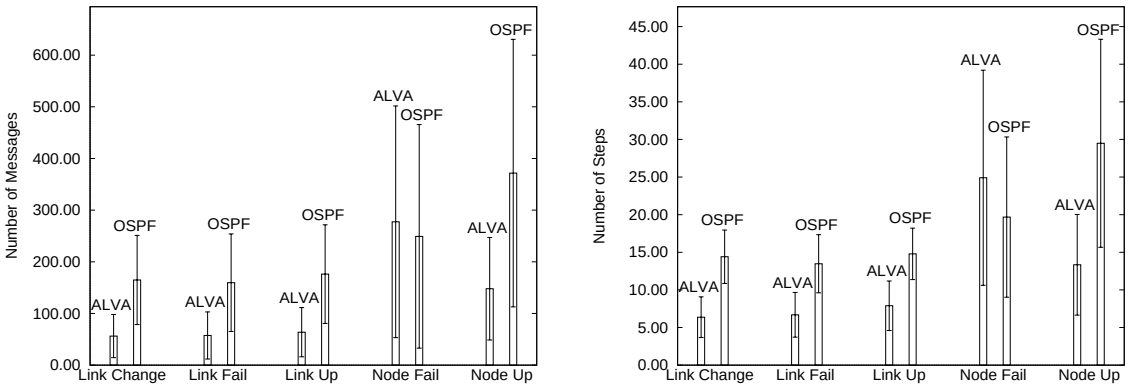

FIGURE 4.7: Topology type 2

fail. As expected, the number of messages required when links change or are established within an area for OSPF is constant. (This is not true for link failures, because some of the failures may disconnect nodes or partition the graph.) The deviation from the mean for such changes in the backbone shows that changes in the backbone cause traffic in the areas in addition to the traffic within the backbone.

In all simulations, ALVA clearly outperforms OSPF when link changes occur, links fail, or new links are established. The only case where OSPF converges with less overhead is when a node fails. In this case, the delete operation in LVA causes ALVA to create slightly more packets than OSPF. However, altogether a router going down and coming back up still creates less traffic in ALVA than in OSPF, because ALVA outperforms OSPF by a wide margin when a new node is established.

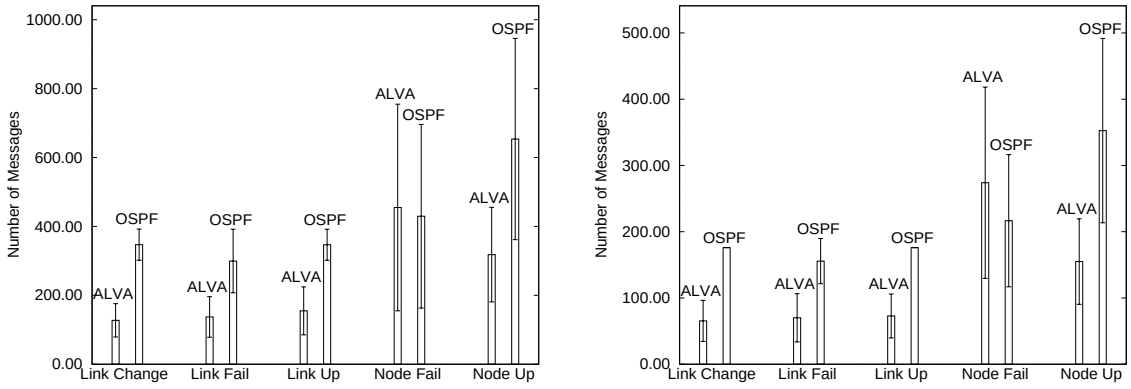

FIGURE 4.8: Topology type 1, backbone (left) and area (right)

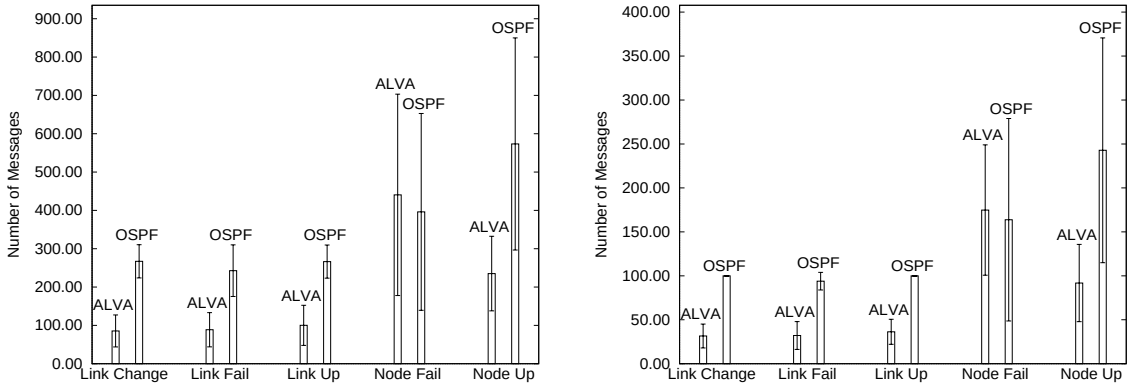

FIGURE 4.9: Topology type 2, backbone (left) and area (right)

Figure 4.10 shows the average size of the topology tables at routers in the backbone and in another area for both types of topologies. It can be seen that routers using ALVA need to keep only about half as many links in their tables on the average when compared to routers OSPF. This is true in particular for backbone routers, which include the border-nodes that run two copies of the topology broadcast (one for the backbone and one for their area.) As the size of the areas grows, this advantage for ALVA becomes even more pronounced. We have also obtained results using flat LVA for larger topologies than the areas shown here. The results obtained that way confirmed the significant savings in the number of links in the tables.

In these simulations we did not address the quality of the paths. Since border nodes in OSPF advertise their actual distance to remote areas rather than just connectivity, paths

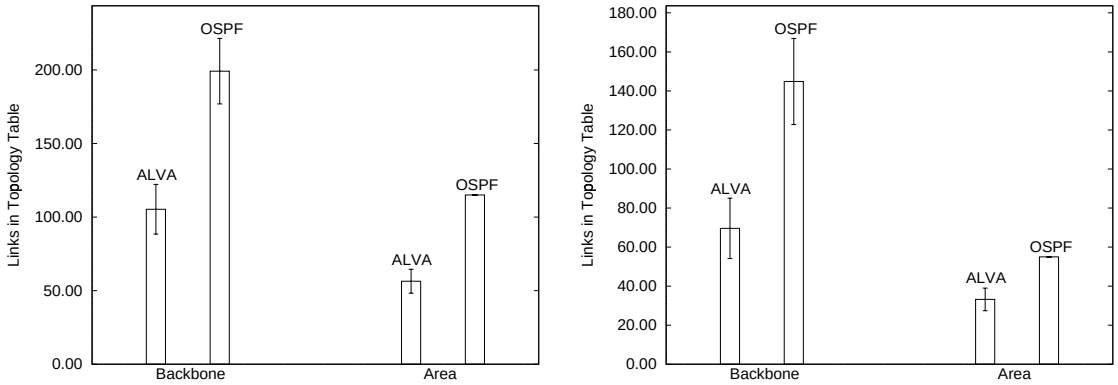

FIGURE 4.10: Size of topology tables. left: topology type 1; right: topology type 2

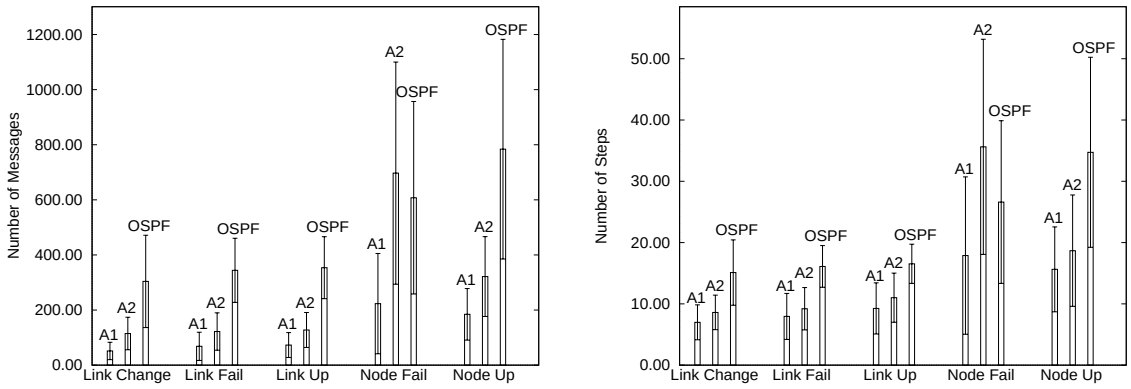

FIGURE 4.11: Topology of type 1 with backbone that can be partitioned (A1: ALVA with backbone area partitioned into three areas; A2: ALVA with contiguous backbone area; OSPF with contiguous backbone area)

derived this way can potentially be shorter. However, we expect border nodes of the same area to be close together and do not expect significant gains when compared to paths obtained by ALVA. On the other hand, ALVA does not require the star configuration forced by OSPF (i.e., using a backbone.)

As an internetwork grows larger, the backbone-based topology required in OSPF forces the backbone to become larger or the areas that connect to it to grow larger. Figure 4.11 illustrates the savings that can be derived with ALVA over OSPF by not requiring a backbone. The topology used for this experiment is of type 1. There are 179 edges in the topology, giving the nodes an average degree of 3.58, with a maximum degree of 9. For the first part of the experiment, the node addresses were chosen such that the backbone was

partitioned into three connected areas; ALVA was used in this scenario. For the second part, the backbone was one contiguous area; both ALVA and OSPF were used in this scenario. The results of this experiment show that the more flexible choice of topologies can widen the margin by which ALVA outperforms OSPF significantly. With the large backbone area required by OSPF partitioned into smaller areas, ALVA outperforms OSPF even when nodes fail.

In addition, in contrast to OSPF, ALVA allows for multiple levels of hierarchy and ALVA does not require a backbone, which means that very large backbones can be broken into smaller areas that provide the same connectivity. These added features make it possible to further reduce communication as well as storage overhead.

4.5 Summary

We have presented a new hierarchical routing algorithm based on link-vector routing and areas for aggregation of routing information. The main idea of LVA is to use link-state information to compute optimal paths but without replicating the complete topology information at every node. This idea has been extended to allow multiple levels of hierarchy. At each level of the hierarchy, partial topology is stored.

The performance of ALVA was compared with that of OSPF. Our simulation results show that, even with only one level of hierarchy, ALVA clearly outperforms OSPF in terms of storage and communication requirements. ALVA does not require a backbone-centered topology, and our simulation experiments illustrate performance advantages gained by allowing arbitrary area-based topologies. In addition, allowing multiple levels of hierarchy makes the new algorithm far more scalable than OSPF. ALVA constitutes the basis for the development of more efficient Internet routing protocols based on link-state information. In particular, new versions of OSPF can be derived in which no backbone is required.

4.6 Appendix: Pseudo Code for ALVA

Variables and Data Structures for ALVA specification

update: tuple $(j, k, c_j^k, sn, type, [h])$
- j, k: origin and destination of link
- c_j^k: cost of link (j, k)
- sn: sequence number
- $type$: ADD or DELETE
- h: head of link, if origin is area address

topology table TT_i at node i with entries:
- j, k: origin and destination of link
- if j local address in same area:
  - c_j^k: cost
  - sn: sequence number
  - list of reporting node
- if j area address
  - c_j^k: connectivity info
  - for each reported head of link
    - sn: sequence number
    - list of reporting nodes
  - if more than one head in list, indicate which one forwarded

TT_i can be subdivided into TT_i^l, where l indicates the level in the hierarchy.

source graph ST_i, new source graph $NewST_i$

Process Update

This procedure processes an update message. If the content of the message caused a change in the topology table, compute new source graph and routing table, and send update packets to neighbors if necessary.

```
procedure update (i, message)
-- Parameters:
-- i: name of node that executes procedure
-- message: message to be processed
begin
    update_topology_table (i, message)
    if updated then
        build_source_graph
        build routing table
        compare_source_graphs (i)
        ST_i = NewST_i
    end if
end update
```

Update Topology Table

```
procedure update_topology_table (i, message)
– – Parameters:
– – i: name of node that executes procedure
– – message: message to be processed
begin
    for all updates in message do
        if local_address ( j ) then
            update as in plain LVA, using table TT_i^0
        else   – – remote address at level l (implies that k is remote address as well)
            if type = ADD then
                if (j,k) ∉ TT_i^l then
                    add link to TT_i^l
                else if h is head of link then
                    if sn > TT_i^l(j,k).h.sn then
                        change link values
                    else if sn = TT_i^l(j,k).h.sn then
                        add reporting node n
                    else if sn < TT_i^l(j,k).h.sn then
                        send correction to n
                    end if
                else
                    add h as head of link
                end if
            else – – type = DELETE
                if h is head of link then
                    if (j,k) ∈ TT_i^l then
                        if sn > TT_i^l(j,k).h.sn then
                            mark h as deleted
                            if no other head then
                                mark link as deleted
                            end if
                        else if sn = TT_i^l(j,k).h.sn then
                            delete reporting node n
                            if no reporting node for h then
                                mark h as deleted
                                if no other head then
                                    mark link as deleted
                                end if
                            end if
                        else if sn < TT_i^l(j,k).h.sn then
                            send correction to n
                        end if
                    else
                        TT_i^l(j,k).h.sn = sn
                    end if
                end if
            end if
        end if
    end for
end update_topology_table
```

Link Change

This procedure is called by the underlying protocol when the cost of a link changed. It changes link information in the topology table at the appropriate hierarchy level; then computes a new source graph and routing table and sends updates to neighbors if necessary.

procedure link_change (i, j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of destination of link
begin
  if local_address (j) **then**
    $TT_i^0(i, j) = (c_i^j, new\_sn, \{i\})$
  else
    $TT_i^0(i, area(j)) = (c_i^j, new\_sn, \{i\})$
  end if
  build_source_graph
  build routing table
  compare_source_graphs (i)
  $ST_i = NewST_i$
end link_change

Link Up

Called by underlying protocol when new neighbor is discovered. Change link information in topology table at appropriate hierarchy level; compute new source graph and routing table and send updates to neighbors if necessary.

procedure link_up (i, j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of destination of link
begin
  $N_i = N_i \cup j$
  if local_address (j) **then**
    $TT_i^0(i, j) = (c_i^j, new\_sn, \{i\})$
  else
    $TT_i^0(i, area(j)) = (c_i^j, new\_sn, \{i\})$
  end if
  build_source_graph
  build routing table
  compare_source_graphs (i)
  $ST_i = NewST_i$
end link_up

Link Down

Called by the underlying protocol when a link failed. Changes the link information in the topology table; removes the destination of the link from all sets of reporting nodes; computes a new source graph and routing table and sends updates to neighbors if necessary.

procedure link_down (i, j)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of destination of link
begin
  $N_i = N_i - j$
  for all $(k, l) \in TT_i$ **do**
    $TT_i^0(k, l).r = TT_i^0(k, l).r - j$
    if $TT_i^0(k, l).r = \emptyset$ **then**
      mark (k, l) as deleted
    end if
  end for
  if local_address (j) **then**
    $TT_i^0(i, j) = (c_i^j, new\_sn, \{i\})$
  else
    $TT_i^0(i, area(j)) = (c_i^j, new\_sn, \{i\})$
  end if
  build_source_graph
  build routing table
  compare_source_graphs (i)
  $ST_i = NewST_i$
end link_down

Compare Source Graphs

This procedure assembles update packets to be sent to neighbors. It uses procedure check_link_in_source_graphs to determine the right type of update.

procedure compare_source_graphs (i)
– – Parameters:
– – i: name of node that executes procedure
begin
  for all $(j, k) \in NewST_i$, $(j, k) \notin ST_i$ **or** $NewST_i(j, k).sn > ST_i(j, k).sn$
      or change in head of link **do**
    check_link_in_source_graphs $(i, j, k,$ ADD)
  end for
  for all $(j, k) \in ST_i$, $(j, k) \notin NewST_i$ **do**
    check_link_in_source_graphs $(i, j, k,$ DELETE)
  end for
  if border-node (i) **then**
    send (inter_area_message)
  send (intra_area_message)
end compare_source_graphs

Check Link in Source Graph

Determines which type of updates need to be sent; converts addresses in updates that are sent across borders to hierarchical form if necessary.

```
procedure check_link_in_source_graphs (i, j, k, type)
– – Parameters:
– – i: name of node that executes procedure
– – j: name of head of link
– – k: name of destination of link
– – type: type of update (ADD/DELETE)
  begin
    if border-node (i) then
      if i = j then
        if not local_address (k) then
          inter_area_message = inter_area_message ∪ (area(i),k,connectivity,sn,type,i)
        end if
        intra_area_message = intra_area_message ∪ (j, k, c_j^k, sn,type)
      else
        if remote_address (j) (level l) then
          inter_area_message =
            inter_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,type,TT_i^l(j,k).h)
          intra_area_message =
            intra_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,type,TT_i^l(j,k).h)
          if change in h and type = ADD then
            inter_area_message =
              inter_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,DELETE,TT_i^l(j,k).h)
            intra_area_message =
              intra_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,DELETE,TT_i^l(j,k).h)
          end if
        else
          intra_area_message = intra_area_message ∪ (j, k,c_j^k,TT_i^l(j,k).sn,type)
          if remote_address (k) (level l)then
            inter_area_message =
              inter_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,type,j)
          end if
        end if
      end if
    else – – interior node
      if remote_address (j) (level l)then
        intra_area_message =
          intra_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,type,TT_i^l(j,k).h)
        if change in h and type = ADD then
          intra_area_message =
            intra_area_message ∪ (j, k,connectivity,TT_i^l(j,k).h.sn,DELETE,TT_i^l(j,k).h)
        end if
      else
        intra_area_message =
          intra_area_message ∪ (j, k,c_j^k,TT_i^l(j,k).sn,type)
      end if
    end if
  end check_link_in_source_graphs
```

Chapter 5

Conclusions

A new method for distributed routing in computer communications networks and internets, LVA, has been introduced. LVA uses link-state information to derive the forwarding tables, keeping partial topology information at every node. The topology database at each node is comprised of the links adjacent to that node and the links used by the neighbors. It has been proven that LVA is correct under different types of routing policies, assuming that a correct mechanism is used for routers to ascertain which updates are recent or outdated.

The performance of LVA has been analyze both analytical and with simulations. The results obtained show that LVA outperforms link-state algorithms as well as distance-vector algorithms based on the distributed Bellman-Ford algorithm. In addition, LVAs have good scaling properties, they scale well with the number of service types and with the number of destinations.

To validate the link-state updates disseminated by LVA or link-state algorithms, sequence numbers are used, and must be drawn from a finite sequence number space. We introduced an algorithm to reset the sequence number assigned to a link that is based on a recursive query-response process and does not need periodic retransmissions or aging, or the establishment of trusted neighbors. This is the first such reset algorithm reported to date.

We have shown that the reset algorithm leads to a correct routing protocol when applied to the selective dissemination of link-state information, and presented a topology broadcast algorithm based on it.

To make link-vector algorithms scale to very large networks, we extended the basic algorithm to an area-based hierarchical routing algorithm. In this algorithm, the internal structure of an area is transparent to routers that are not part of the area. The basic idea of LVA is extended to allow for multiple levels of hierarchy. At each level of hierarchy, a partial view of the connectivity between areas is stored.

An obvious question for future research is the performance of ALVA with multiple levels of hierarchy. Because the simulation tool limited our experiments to graphs with 100 nodes, we were not able to fully analyze the scalability of ALVA in this thesis.

One important area for future research is the optimization of the local computations. A major drawback of link-state protocols is that they need to run a computationally expensive shortest-path algorithm at every node. Although the smaller topology table size in LVA improves the running time for the shortest-path algorithm, the local computations are still far more complex than those needed in distance-vector algorithms. Improvements may be possible by taking advantage of the structure of the topology table and the source graph. An ideal solution for the local computations would provide for incremental changes in the source graph, rather than a complete rebuilding after any change.

Multimedia and teleconferencing applications using the Internet create a demand for quality of service (QoS) guarantees, such as a guaranteed bandwidth and delays. To provide such guarantees, it is necessary to avoid congestion in the network. Today's routing algorithms are poorly integrated with congestion control [Kou97]. To be able to support QoS, congestion control and avoidance need to be better integrated with the routing decisions. Extending LVA to provide multiple loop-free paths to the same destination, as well as computing such paths based on multiple constraints (e.g., delay, capacity, jitter) are research topics that need to be addressed. Another important topic in this context

is investigating the support of multicast routing as part of LVA, i.e., extending LVA to accommodate multicasting like DVMRP extends the distributed Bellman-Ford algorithm.

Bibliography

[AGB94] R. Albrightson, J.J. Garcia-Luna-Aceves, and J. Boyle. EIGRP–a fast routing protocol based on distance vectors. In *Proc. Networld/Interop 94*, Las Vegas, Nevada, May 1994.

[APV94] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded. In *Proceedings of IEEE INFOCOM '94 Conference*, volume 2, pages 776–783, Toronto, Ont., Canada, 12-16 June 1994.

[BG92] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1992.

[BG94] J. Behrens and J.J. Garcia-Luna-Aceves. Distributed, scalable routing based on link-state vectors. In *Proceedings ACM SIGCOMM '94 Conference*, pages 136–147, London, UK, August 31 to September 2 1994.

[BG97] J. Behrens and J.J. Garcia-Luna-Aceves. Fast dissemination of link states using bounded sequence numbers with no periodic updates or age fields. In *International Conference on Distributed Computing Systems (ICDCS'97)*, Baltimore, Maryland, May 1997.

[Bos92] L. Bosack. Method and apparatus for routing communications among computer networks. U.S. Patent assigned to Cisco Systems, Inc., Menlo Park, CA, February 1992.

[BT89] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[CCS95] I. Castineyra, J. N. Chiappa, and M. Steenstrup. The Nimrod Routing Architecture. Internet Draft, January 1995.

[Cis97] Cisco Systems, http://www.cisco.com/univercd/home/home.htm. *Cisco Online Documentation: Technology Information*, 1997.

[CRKG89] C. Cheng, R. Riley, S. Kumar, and J.J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *Computer Communications Review*, 19(4):224–236, September 1989.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[ERH92] D. Estrin, Y. Rekhter, and S. Hotz. Scalable inter-domain routing architecture. *Computer Communications Review*, 22(4):40–52, Oktober 1992.

[EST93] D. Estrin, M. Steenstrup, and G. Tsudik. A protocol for route establishment and packet forwarding across multidomain internets. *IEEE/ACM Transactions on Networking*, 1(1):56–70, February 1993.

[Gaf87] E. Gafni. Generalized scheme for topology-update in dynamic networks. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science No. 312*, pages 187–196. Springer-Verlag, 1987.

[Gar86] J.J. Garcia-Luna-Aceves. A fail-safe routing algorithm for multihop packet-radio networks. In *Proc. of IEEE INFOCOM '86*, Miami, Florida, April 1986.

[Gar88] J.J. Garcia-Luna-Aceves. Routing management in very large-scale networks. *Future Generation Computing Systems (FGCS)*, 4(2):81–93, September 1988.

[Gar89] J.J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. In *Proc. SIGCOMM '89 Symposium Communications Architectures and Protocols*, pages 212–223, September 1989.

[Gar92] J.J. Garcia-Luna-Aceves. Reliable broadcast of routing information using diffusing computations. In *Proc. IEEE Globecom '92*, volume 1, pages 615–621, Orlando, Florida, December 1992.

[Gar93] J.J. Garcia-Luna-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1):130–141, February 1993.

[GB95] J.J. Garcia-Luna-Aceves and J. Behrens. Distributed, scalable routing based on vectors of link states. *IEEE Journal on Selected Areas in Communications*, 13(8):1383–95, October 1995.

[GM97] J.J. Garcia-Luna-Aceves and S. Murthy. A path finding algorithm for loop-free routing. *IEEE/ACM Transactions on Networking*, 5(1):148–160, February 1997.

[GR97] R. Govindan and A. Reddy. An analysis of internet inter-domain topology and route stability. In *Proc. IEEE INFOCOM '97*, Kobe, Japan, April 7-11 1997.

[GZ94] J.J. Garcia-Luna-Aceves and W.T. Zaumen. Area-based, loop-free internet routing. In *Proc. IEEE INFOCOM '94*, pages 1000–1008, Toronto, Canada, June 1994.

[Hed88] C. Hedrick. Routing information protocol. RFC 1058, Network Information Center, SRI International, Menlo Park, CA, June 1988.

[HS89] P.A. Humblet and S.R. Soloway. Topology broadcast algorithms. *Computer Networks and ISDN Systems*, 16(3):179–186, January 1989.

[Hui95] C. Huitema. *Routing in the Internet.* Prentice Hall PTR, Englewood Cliffs, NJ, 1995.

[Hum91] P. Humblet. Another adaptive distributed shortest path algorithm. *IEEE Transactions on Communications*, 39(6):995–1003, June 1991.

[ISO] International Standards Organization. *Protocol for Exchange of Inter-domain Routing Information among Intermediate Systems to Support Forwarding of ISO 8473 PDUs*, ISO/IEC/JTC1/SC6 CD10747.

[ISO89] International Standards Organization. *Intra-Domain IS-IS Routing Protocol*, ISO/IEC JTC1/SC6 WG2 N323, September 1989.

[Jaf84] J.M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[JM82] J.M. Jaffe and F.M. Moss. A responsive routing algorithm for computer networks. *IEEE Transactions on Communications*, 30(7):1758–1762, July 1982.

[Kam76] F. Kamoun. *Design Considerations for Large Computer Communication Networks.* PhD thesis, University of California, Los Angeles, 1976.

[KK77] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks: Performance evaluation and optimization. *Computer Networks*, 1(3):155–174, 1977.

[Kou97] D. Kourkouzelis. Multipath routing using diffusing computations. Master's thesis, University of California, Santa Cruz, March 1997.

[KZ89] A. Kanna and J. Zinky. A revised ARPANET routing metric. In *Proc. SIGCOMM '89*, pages 45–56, September 1989.

[LR91] K. Lougheed and Y. Rekhter. Border Gateway Protocol 3 (BGP-3). RFC 1267, SRI International, Menlo Park, CA, October 1991.

[McQ74] J. McQuillan. Adaptive routing algorithms for distributed computer networks. BBN Rep. 2831, Bolt Beranek and Newman Inc., Cambridge MA, May 1974.

[MG97] S. Murthy and J.J. Garcia-Luna-Aceves. Loop-free internet routing using hierarchical routing trees. In *Proc. IEEE INFOCOM '97*, Kobe, Japan, April 7–11 1997.

[Moy94] J. Moy. OSPF Version 2. RFC 1583, Network Working Group, March 1994.

[Mur96] S. Murthy. *Routing in Packet-Switched Networks Using Path-Finding Algorithms.* PhD thesis, University of California, Santa Cruz, September 1996.

[Nov94] Novell. *NetWare Link Services Protocol (NLSP) Specification, Revision 1.0.* Novell, Inc., San Jose, CA 95131, 2nd edition, February 1994.

[Per83] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7(6):395–405, 1983.

[PVL92] R. Perlman, G. Varghese, and A. Lauck. Reliable Broadcast of Information in a Wide Area Network. US Patent 5,085,428, February 1992.

[Rek93] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). *Internetworking: Research and Experience*, 4(2):61–80, June 1993.

[RF91] B. Rajagopalan and M. Faiman. A responsive distributed shortest-path routing algorithm within autonomous systems. *Internetworking: Research and Experience*, 2(1):51–69, March 1991.

[RHE94] Y. Rekhter, S. Hotz, and D. Estrin. Constraints on forming clusters with link-state hop-by-hop routing. Technical Report USC-CS-93-536, University of Southern California, 1994.

[Rid84] G.G. Riddle. Message routing in a computer network. U.S. Patent assigned to AT&T Bell Telephone Laboratories, Inc., Patent Number 4,466,060, August 1984.

[RT83] C.V. Ramamoorthy and W.T. Tsai. An adaptive hierarchical routing algorithm. In *Proc. COMPSAC 83*, pages 93–104, Chicago, IL, USA, November 1983. IEEE.

[SADM95] A.U. Shankar, C. Alattinoglu, K. Dussa-Zieger, and I. Matta. Transient and steady-state performance of routing protocols: Distance-vector versus link-state. *Internetworking: Research and Experience*, 6(2):59–87, June 1995.

[Ste95] M. Steenstrup. *Routing in Communications Networks.* Prentice Hall, Englewood, Cliffs, NJ, 1995.

[Tho96] S.A. Thomas. *IPng and the TCP/IP Protocols: implementing the next generation internet.* John Wiley & Sons, New York, 1996.

[TRTN89] W.T. Tsai, C.V. Ramamoorthy, W.K. Tsai, and O. Nishiguchi. An adaptive hierarchical routing protocol. *IEEE Transactions on Computers*, 38(8):1059–1075, Augst 1989.

[Tsu88] P.F. Tsuchiya. The landmark hierarchy: A new hierarchy for routing in very large networks. *Computer Communications Review*, 18(4):43–54, 1988.

[ZCB96] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings IEEE INFOCOM '96*, volume 2, pages 594–602, San Francisco, CA, March 26–28 1996.